

Evaluating and Comparing Memory Error Vulnerability Detectors

Yu Nong^a, Haipeng Cai^{a,✉}, Pengfei Ye^a, Li Li^b and Feng Chen^c

^aSchool of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, 99164

^bFaculty of Information Technology, Monash University, Melbourne, Australia, 3800

^cDepartment Of Computer Science, University of Texas, Dallas, TX, 75080

ARTICLE INFO

Keywords:

Memory error vulnerability
vulnerability detection
open-source tools
code analysis
empirical evaluation
comparative study
benchmark selection

ABSTRACT

Context: Memory error vulnerabilities have been consequential and several well-known, open-source memory error vulnerability detectors exist, built on static and/or dynamic code analysis. Yet there is a lack of assessment of such detectors based on rigorous, quantitative accuracy and efficiency measures while not being limited to specific application domains.

Objective: Our study aims to assess and explain the strengths and weaknesses of state-of-the-art memory error vulnerability detectors based on static and/or dynamic code analysis, so as to inform tool selection by practitioners and future design of better detectors by researchers and tool developers.

Method: We empirically evaluated and compared five state-of-the-art memory error vulnerability detectors against two benchmark datasets of 520 and 474 C/C++ programs, respectively. We conducted case studies to gain in-depth explanations of successes and failures of individual tools.

Results: While generally fast, these detectors had largely varied accuracy across different vulnerability categories and moderate overall accuracy. Complex code (e.g., deep loops and recursions) and data (e.g., deeply embedded linked lists) structures appeared to be common, major barriers. Hybrid analysis did not always outperform purely static or dynamic analysis for memory error vulnerability detection. Yet the evaluation results were noticeably different between the two datasets used. Our case studies further explained the performance variations among these detectors and enabled additional actionable insights and recommendations for improvements.

Conclusion: There was no single most effective tool among the five studied. For future research, integrating different techniques is a promising direction, yet simply combining different classes of code analysis (e.g., static and dynamic) may not. For practitioners to choose right tools, making various tradeoffs (e.g., between precision and recall) might be inevitable.

1. Introduction

Cybersecurity threats are considerably attributed to software vulnerabilities. In response, various kinds of techniques have been developed to detect these vulnerabilities, yet there is a lack of understanding of their strengths and limitations. Many of the existing relevant studies are limited to technical discussion and qualitative assessment [23, 1, 39], domain-specific scopes (e.g., SQL injection vulnerabilities [41, 2, 31, 19]), and/or incomplete comparisons (e.g., only concerning the numbers of vulnerabilities found rather than computing and comparing precision and recall) [7, 6, 32, 3]. In [5], benchmarking metrics were proposed yet not applied.

A few benchmarking studies (with precision and recall quantified against ground truth) exist [4, 14, 37], yet they targeted commercial tools exclusively. Without knowing the technical details of these tools, it is difficult to explain their performance differences or insufficiency, which is necessary for distilling insights into the design of future advanced techniques. As it stands, there has been no published comparative evaluation of open-source code-analysis-based vulnerability detectors that (1) uses the

same benchmark datasets with ground truth to quantitatively assess precision and recall, (2) provides insights into the variations in detection performance, and (3) offers actionable recommendations for future detector design and development. Performing such a comparative evaluation is key to understanding where we are with such detection techniques and how we may improve in this domain.

To fill this gap, we conducted an extensive study of five state-of-the-art, open-source memory error vulnerability detectors [27, 9, 35, 40, 20]. While these detectors may also be built on data-driven (e.g., machine/deep learning) techniques, we start with focusing on the techniques that are primarily based on (static, dynamic, or hybrid) code analysis, given that the most commonly adopted detectors are of the latter class. We chose to target memory error vulnerabilities (i.e., those compromising memory safety) because (1) they represent a dominant class of vulnerabilities in modern C/C++ software, (2) C/C++ has been by far the most vulnerable language [33] yet also the language in which many critical software systems are written, and (3) memory error vulnerability detectors have seen promising practical adoption. Our goal is to provide an evaluation and comparison of these detectors hence the types of techniques underlying them, so as to understand potential gaps in their detection performance. To that end, our study design has the following key elements.

First, we measured the detection performance in terms

[✉]Corresponding author

✉ yu.nong@wsu.edu (Y. Nong); haipeng.cai@wsu.edu (H. Cai); pengfei.ye@wsu.edu (P. Ye); li.li@monash.edu (L. Li); Feng.Chen@utdallas.edu (F. Chen)
ORCID(s): 0000-0002-5224-9970 (H. Cai); 0000-0003-2990-1614 (L. Li); 0000-0002-4508-5963 (F. Chen)

of rigorously computed accuracy metrics (precision, recall, and F1 score). Second, we used two benchmark datasets that are different in multiple ways (e.g., objective, size, distribution of vulnerable and non-vulnerable samples, closeness to real-world programs) while applying them to the same five detectors. We chose to do so in order to gain a more complete picture of the performance of these tools through a more comprehensive comparative evaluation. Third, we ensured that both datasets only include memory error vulnerabilities, in accordance with our scope of evaluating detectors targeting memory related vulnerabilities. Thus, from existing datasets we selected samples that are relevant to our study. We do not intend to build a new benchmark or benchmarking approach.

We separately set up, configured, and ran five vulnerability detectors against each of the two datasets with different approaches according to the different ways they work (see Figure 1). Then, we collected the vulnerability detection results, and computed the statistics on the accuracy and efficiency (time cost and memory usage) of these detectors. We considered precision, recall, and F1 score as our accuracy metrics, computed against the vulnerability ground truth available with the two datasets.

With these chosen benchmark datasets and detectors, we sought to answer five research questions:

- **RQ1:** *How effective are these detectors in terms of accuracy?* Previous relevant works were mainly based on counting the number of vulnerabilities the detectors found against benchmarks without respective ground truth. Thus, the accuracy in terms of precision, recall and F1 score was not fully measured. In our study, we ran each of the chosen detectors against each sample and collected the detection results (e.g., vulnerable code lines), from which we computed the tool accuracy in terms of precision, recall, and F1 score, separately for each of the vulnerability categories covered by each of the two benchmark datasets.
- **RQ2:** *How do these detectors compare in terms of their accuracy?* As mentioned, a key aim of our study is to understand why certain techniques perform well while others do not, so as to distill new understandings and knowledge about the design of memory error vulnerability detection, at least in the realm of open-source tools. Thus, we compared accuracy in terms of F1 score among the chosen detectors. Since we intend to explain the performance of current techniques and their differences in that regard according to their technical nature, it is essential to evaluate the performance differences with statistical evidence. Such evidence would provide necessary confidence about the validity of ascribing performance gaps to technical differences. As a result, we conducted statistical analyses, computing the statistical significance and effect size of performance differences, to assess the

strength of the confidence.

- **RQ3:** *How efficient are the detectors in terms of their time costs and memory usage for detecting vulnerabilities?* Efficiency is another important metric for evaluating the performance of a detector. A technique costing too much time and memory for detecting vulnerabilities in a program may encounter adoption barriers in practice. Thus, we assessed and compared the efficiency of the detectors by measuring their time costs and memory usage for detecting each of the program samples. We also evaluated the scalability of the detectors by investigating the relationship between the efficiency (i.e., time costs and memory usage) and the program size (i.e., lines of code), so as to identify any potential efficiency challenges faced by these state-of-the-art techniques.
- **RQ4:** *Why did some of the detectors fail with certain instances of vulnerabilities?* Several previous works compared commercial vulnerability detectors in terms of precision, recall and F1 score. However, there was no deep investigation into those detectors regarding the underlying causes of large performance gaps or failure cases. Our preliminary study [30] suffers the same drawback. Yet understanding those causes is an important means for deriving insights about developing future vulnerability defense techniques. Thus, we conducted a set of in-depth case studies with respect to notable failure and low-performing cases, dissecting the underlying causes. We also provided actionable insights into how these failures could be addressed.
- **RQ5:** *How did the benchmark selection impact the evaluation results?* The performance of the chosen detectors can be different just because of the different selection of benchmark datasets. Intuitively, many factors of the datasets (e.g., size, composition, etc.) might affect the evaluation results. Studying such effects can help understand the reliability of the empirical assessment hence identify potential biases. Thus, we compared the overall performance of the five detectors against the two datasets, from which we analyzed the impact of benchmark selection.

Among other findings, our study revealed:

1. The accuracy of these detectors varied widely, due to differences among various vulnerability categories and different underlying technical design choices. Meanwhile, the results suggested that the accuracy of these detection tools can be further improved.
2. Incorporating static analysis in vulnerability detection tended to bring substantial accuracy benefits, especially compared to purely dynamic approaches, mainly due to the low recall of the latter.

3. Both complex code structure (e.g., deep loops and recursions) and sophisticated data structures (e.g., deeply embedded linked lists) constituted significant barriers for effective memory error vulnerability detection with the studied tools. Also, vulnerabilities in real-world applications tended to be more challenging to detect than those generated artificially.
4. In general, it was difficult for any detector to achieve both high precision and high recall at the same time. Design choices made in a technique (e.g., DRMEMORY) may favor precision (recall) but compromise recall (precision) at the same time.
5. Among the studied detectors, those based on hybrid analysis were not always more accurate than those based on purely static or dynamic analysis. Likewise, purely dynamic detectors did not always outperform the purely static detector, and vice versa.
6. The detectors were generally efficient, taking no more than 5 minutes and no more than 2.5GB memory at maximum for a program sample, despite timeout cases with some detectors against certain program samples. The time costs and memory usage had moderate and non-monotonic variations for different program sizes, indicating that the detectors were scalable for our chosen datasets.
7. The selection of benchmarks had noticeable impact on the performance measurements of the studied detectors. The highest accuracy (F1 score) achieved by any of these detectors overall was 87% against the *Software-Analysis-Benchmark* dataset, while the number was 72% against the *SV-Benchmark* dataset. This difference can be attributed to the fact that the former is noticeably less complex in code and data structures than the latter.

In sum, our main goal and contribution is to provide, through a comparative evaluation, empirical results for understanding where current vulnerability defense techniques are. Focusing on memory error vulnerabilities and techniques detecting them based on code analysis, our study complements existing peer work in terms of scope and depth. Also, our results shed light on how techniques of varied nature perform differently and why. Based on our empirical findings, we further distilled key lessons learned and provided recommendations for both researchers and practitioners in the field of software vulnerability analysis. We have released the code and datasets used in our study to facilitate reproduction and reuse, as found [here](#).

2. Background

This section provides key background materials necessary for understanding the rest of the paper.

2.1. Software Vulnerabilities

Software vulnerabilities are weaknesses in software that are exploitable for malicious purposes [38]. They may result in consequences like information leak, software

crashes, and data tampering. Vulnerabilities are different from bugs. A software bug can be any defect, while a vulnerability is a bug that poses security threats. Generally whether a bug is a vulnerability or not depends on specific program contexts. It is possible that one type of software bugs constitutes vulnerabilities in one program, while they do not in another. In our study, to investigate potential vulnerabilities more comprehensively, we treated all types of bugs that might be used for malicious purposes as vulnerabilities, regardless of their program contexts and practical consequences.

2.2. Loop Unwinding

Loop unwinding, or loop unrolling, is a technique used for optimizing the execution efficiency of programs. It rewrites the loop statements in programs into repeated sequences of independent statements, so that some offsets of array variables can be pre-calculated and built into the binary code instructions directly hence the run time of arithmetic computation can be reduced [29]. Since loop unwinding can be used to transform program statements into related plain statements, it is also widely used in formal verification techniques, especially bounded model checking [22, 20, 25]. Control flow graphs (CFGs) where loops are eliminated by loop unwinding can be easily used to generate conjunctive normal forms (CNFs) that can be solved by SAT solvers for property verification. However, because of the limited memory and computation resources, loop unwinding cannot be used to comprehensively verify programs with too many or endless loops and recursions.

2.3. Shadow Memory

Shadow memory is a technique that tracks the information about memory and registers as a program is running, by mapping individual bits or bytes in the main memory and registers into shadow bytes. This technique is widely utilized for dynamically detecting memory related vulnerabilities, such as invalid access, memory leak, double free, and use after free. More specifically, it can be used to track allocation and deallocation operations in heap and stack memory to detect uninitialized access [27, 9, 35]. It can also be used to find data from untrusted sources and detect unsafe uses of untrusted data [28, 10, 15]. Shadow memory can be implemented in different ways that have different effectiveness, efficiency and robustness [26] implications. For instance, all of the four dynamic detectors considered in our study use shadow memory for detecting memory error vulnerabilities.

3. Methodology

We start with an overview of the design of our study. Then, we describe the benchmark datasets and vulnerability detectors chosen, procedure of our study, and metrics and measures considered.

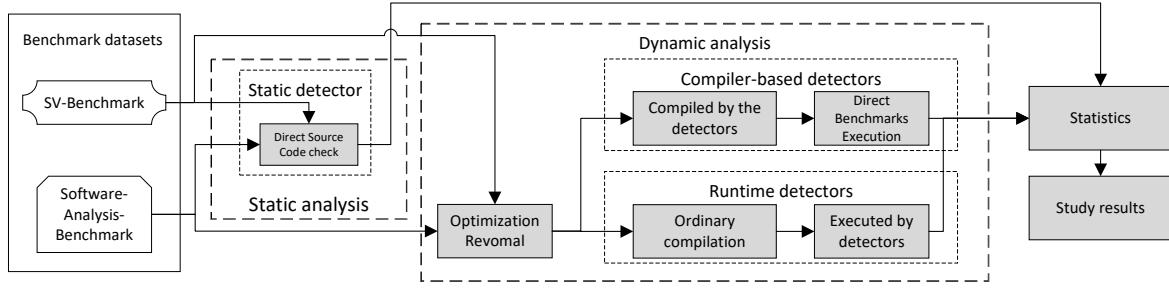


Figure 1: An overview of the overall process flow of our study.

3.1. Process Overview

Figure 1 depicts the process flow of our study. We selected samples from the *Software-Analysis-Benchmark* and *SV-Benchmark* datasets that are most relevant to memory error vulnerabilities, on which our study is focused. While the chosen *detectors* all focus on detecting memory error vulnerabilities, they implemented different techniques in different ways. Thus, for tools of different workings and configurations, we took different approaches to perform our experiments.

For *static tools* (i.e., those that are based on purely static code analysis and scan code without recompiling the code), we developed and ran scripts so that the tools could run against every program sample automatically. Then, we manually checked the outputs to compute statistics of results including accuracy and efficiency (i.e., run time and memory usage).

For tools that are based on integrating the analysis mechanisms into subject programs via compilation (referred to as *compiler-based tools*), we first removed all optimization flags before compiling so that the capabilities of the tools can be tested genuinely. Next, we built each program sample with each tool, reported the compiling time, and ran the rebuilt program sample to obtain results through study scripts. Then, we computed relevant statistics of the results as for the static tools.

For *runtime tools* (i.e., those that only detect vulnerabilities in executables), the process is similar to what we followed for the compiler-based tools, except for building the program samples with an ordinary compiler and running the programs along with the detectors.

With the results on accuracy and efficiency of individual tools, we computed the statistical significance between each pair of the detectors we studied, so as to assess the differences among them. Then, we picked representative program samples to perform in-depth case studies in order to further explain the reasons why certain tools succeeded or failed in some situations, hence to gain a deeper understanding about the performance of the underlying techniques.

3.2. Benchmark Datasets

We used two disparate datasets as mentioned earlier as our benchmarks. We elaborate each below.

The first was formed by samples from the *Software-Analysis-Benchmark* introduced in [37], which includes 638 positive (i.e., vulnerable) samples and 638

Table 1

Categories of chosen samples in *Software-Analysis-Benchmark*

Category	#Samples	#Positives	#Negatives
Dynamic Buffer Overrun	64	32	32
Dynamic Buffer Underrun	78	39	39
Static Buffer Overrun	108	54	54
Static Buffer Underrun	26	13	13
Stack Overflow	14	7	7
Stack Underrun	14	7	7
Invalid Memory Access Already Freed Area	34	17	17
Cross Thread Stack Access	12	6	6
Double Release	12	6	6
Double Free	24	12	12
Free Non-Dynamically Allocated Memory	32	16	16
Free Null Pointer	28	14	14
Data Overflow	50	25	25
Data Underflow	24	12	12
Total	520	260	260

corresponding negative (i.e., non-vulnerable) samples in C/C++—each sample came with a vulnerable version and a corresponding non-vulnerable version (i.e., with the vulnerability fixed). While the authors originally curated the benchmark for evaluating static analysis detectors, we intended to see if and how well the vulnerabilities can be detected by detectors based on purely dynamic or hybrid analysis as well. All of the samples in this dataset covered 51 categories of errors (e.g., static/dynamic buffer overrun, stack overflow, data overflow, etc.), out of which we chose 14 categories that are most relevant to memory safety. Accordingly, we obtained 260 positive samples and the 260 associated negative samples, as summarized in Table 1. These 520 C/C++ programs formed the first dataset actually used in our study. Each of these programs came with the vulnerability ground truth, which enabled our accuracy computation. All of these samples are artificially created and generally simple in terms of code complexity.

For a more comprehensive and extensive comparative assessment, we have considered another dataset. This second dataset was formed by samples chosen from *SV-Benchmark*, as previously used for the 2019 edition of *SV-COMP* [8], a competition event that promotes the invention and improvement of software verification. The dataset includes a total of 10,522 samples written in C/C++ in 6 categories for different verification tasks, with each category including several sub-categories. We chose 5 of the sub-categories from the *MemSafety* category, and the *Systems-BusyBox-MemSafety* sub-category from the *SoftwareSystems* category, because these are about memory

Table 2Categories of chosen samples in *SV-Benchmark*

Category	#Samples	#Positives	#Negatives
MemSafety_Arrays	71	22	49
MemSafety_Heap	180	93	87
MemSafety_Linkedlists	103	27	76
MemSafety_MemCleanup	32	32	0
MemSafety_Other	48	20	28
Systems-BusyBox-MemSafety	40	4	36
Total	474	198	276

safety hence closely relevant to our study focus. As a result, we selected 198 positive and 276 negative (not all corresponding to the positive ones) program samples falling in 6 memory error vulnerability categories, as summarized in Table 2. These 474 program samples were contributed by 24 different individuals or organizations from academia or industry. While most of the program samples were artificially generated, 55 of them (including 16 positive and 39 negative ones) were from real-world projects. These 55 had the levels of complexity in algorithmic logic and code/data structures that are representative of real-world applications.

In certain samples, there are some undefined functions that express special information that is difficult to capture with the C language. For software verification tools, such samples are configured to assume that those functions have been implemented in certain ways (e.g., by the run-time environment of the competition). We do not use the competition platform, though. Thus, to allow our tools to run on those samples, we defined such functions with the implementation assumed by the competition platform. The definitions are shown as follows.

- **_VERIFIER_error():** This is for checking (un)reachability as by verification tools. It is not relevant to our experiment. We defined it as:
`void __Verifier_error() { abort(); }`
- **_VERIFIER_assume(expression):** In the function, if the expression is evaluated to be 0, then the function loops forever, otherwise the function returns as normal. We defined it as:
`void __Verifier_assume(int expression)
{ if (!expression) { while(1); } return; }`
- **_VERIFIER_nondet_X():** The function is assumed to return a nondeterministic value. *x* is the type of the return value, such as *bool*, *char*, *int*, *double*, etc. For these functions, we return a random value of type *x*.
`_Verifier_nondet() {}`
- **_VERIFIER_atomic_begin():** This is for modeling atomic execution statements in a multi-threaded run-time environment. In our study, we define it as an empty function:
`_Verifier_atomic_begin() {}`

3.3. Detectors

For our study, we selected the following five state-of-the-art, open-source memory error vulnerability detectors.

CBMC [20] is a bounded model checker that automatically verifies security properties such as memory (e.g., array bounds and pointer use) safety in a C program through validating assertions against property violations. Specifically, it works for a given C program as follows:

1. The property generator in CBMC inserts assertions to the program, which allow CBMC to locate 6 types of bugs: buffer overflow, pointer safety, memory leak [24], dividing by zero, not a number, and arithmetic overflow.
2. CBMC converts the program into intermediate representation (IR) by unrolling iterations and function recursions, so that the IR has forward control flows only. Then, based on the IR, a control flow graph (CFG) is constructed.
3. On the CFG, CBMC traverses all the paths which reach each of the assertions. Then, CBMC symbolically executes each path and builds a CNF (conjunctive normal form) formula for verification. The generated CNFs are verified via an existing high-performance SAT solver. If a CNF is not boolean satisfiable, CBMC reports a potential bug for the respective path.

However, CBMC cannot prove the correctness for programs with unbounded loops. It allows users to define the unwinding bound and depth so that the verification can be terminated within a finite time and resource usage. In our study, we set the unwinding bound and depth to 300 so that most program samples can be terminated with a reasonable time and resource usage.

VALGRIND [27] is a heavy-weight dynamic binary instrumentation framework that supports a group of dynamic binary analysis tools to implement shadow memory for analyzing machine code at runtime. In this study, we used the most popular VALGRIND-based tool *Memcheck* [36], a run-time bit-precise memory error detector, to assess the performance of VALGRIND. VALGRIND/*Memcheck* detects memory errors in a given executable via the following main steps:

1. After the VALGRIND core and *Memcheck* start, the executable is loaded and disassembled into a tree IR.
2. Following the *disassemble-and-resynthesize* (D&R) principle, the tree IR is converted into a flat IR. Then, the instrumentation IR (i.e., analysis code) is inserted for each of the statements in the original IR. These IR statements map each of the memory bits used in the program into a *shadow memory*, and trace and modify the status of each memory bit (i.e., whether it is *defined* for a valid use) based on the behaviors of the original IR statements.
3. The flat IR is optimized and converted back to tree IR and executable machine code.

4. The VALGRIND core starts the program and runs it in a *guest* CPU which is simulated by the core.
5. Through the instrumented code and the *shadow memory*, each of the (a) memory accesses, (b) conditional jumps, and (c) system calls is checked against whether the access to or operation on relevant memory bits is invalid. If so, a memory error would be reported.

DRMEMORY [9] is a cross-platform memory checking tool that dynamically detects memory errors in binary executable programs running on Windows and Unix-like operating systems. Built on DynamoRIO (an open-source dynamic binary instrumentation platform), DRMEMORY works in following main steps:

1. It loads the binary executable and then (dynamically) instruments the machine code directly, following the *copy-and-annotate* (C&A) principle.
2. Dynamic library calls are *wrapped* so that *redzones* (i.e., safety regions for avoiding and detecting invalid accesses to adjacent allocations) are added between dynamic memory regions, to track heap access.
3. The (instrumented) executable is launched within DRMEMORY. Similar to VALGRIND, DRMEMORY allocates shadow memory for each of the memory addresses used in the original program. Yet unlike VALGRIND which supports bit-precise mapping, DRMEMORY only supports byte-precise mapping. Each of the bytes used in the original memory is mapped into a 2-bit shadow memory area which has three states: unaddressable, uninitialized, and defined. The instrumentation inserts code that traces and modifies the shadow memory based on the behaviors of the original code.
4. Any access to unaddressable memory would cause reporting errors. For uninitialized memory, only accesses related to conditional jumps and system calls would cause error reports.
5. To accommodate C++ code, DRMEMORY only reports memory leaks in the middle of the heap, since the dynamic memory allocated with `new[]`, memory for modeling multiple inheritance, and memory for `std::string` objects have headers that the respective pointers do not point to.

ADDRESSSANITIZER [35] is a compiler-based memory error checker for C/C++ programs that is built on LLVM [21] and can find out-of-bounds access to heap, stack, and global objects, as well as use-after-free errors. It does so for a given C/C++ program by the following steps:

1. It recompiles the program while performing source-code-level (static) instrumentation.
2. Unlike DRMEMORY which *wraps* dynamic library function calls, ADDRESSSANITIZER *replaces* them with specialized implementations. Similarly, each dynamic memory region would have a *redzone* before and after it, which is unaddressable.

3. The instrumented code is started directly on the OS (rather than by an executor like VALGRIND or DRMEMORY). At runtime, the code inserted during the instrumentation builds a shadow memory, directly mapping every 8 bytes to 1 byte (no mapping table is required as by VALGRIND and DRMEMORY).
4. The mapped shadow address stores the status of each 8-byte memory block. A status of 0 indicates that the whole memory block is unaccessible; a negative-number status indicates that the whole block is accessible; and a status encoded by a positive number N indicates that the first N bytes of the block are accessible. The code inserted in instrumentation traces and modifies the shadow memory based on the behaviors of the original code. If there is any access to an invalid memory address, the program reports an error and aborts. The analysis combines static program information obtained at compile-time and shadow memory maintained at runtime, representing a *hybrid* analysis for vulnerability detection.

MEMORYSANITIZER [40] is a fast compiler-based dynamic memory error detector that focuses on detecting use of uninitialized memory in C and C++ programs with reduced time and memory overhead. It works in the following major steps:

1. Like ADDRESSSANITIZER, it first recompiles the given program and performs static instrumentation. More specifically, the program is first converted into LLVM IR. Then, MEMORYSANITIZER instruments the IR and inserts shadow memory code to the program for bit-to-bit mapping.
2. The instrumented program is started directly on the OS. During the execution, each of the bits in the original program's memory space is mapped to a shadow memory by simply applying an OR operation with a `ShadowMask` constant. The shadow memory traces whether the original memory is initialized.
3. Based on the instrumentation at compile-time, each of the operations on the memory should have a corresponding operation on the shadow memory. Copying of uninitialized memory would result in the propagation of uninitialized status bits in the shadow memory. However, to achieve higher efficiency, instead of implementing all operations in the shadow memory to model the corresponding operations in the original memory, MEMORYSANITIZER implements approximate propagation by applying simple bit shifts and bit logic operations to trace the memory without yielding too many false positives and false negatives.
4. Similar to other dynamic detectors, any access to a memory address that the shadow memory indicates as uninitialized would result in an error report.

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, MEMORIESANITIZER), 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 dynamic approaches because, among the techniques we surveyed, most of those for which we can find publicly available tools were dynamic. In addition to these source code based analysis approaches, software vulnerability detection has also been addressed through data mining [18, 16] and machine learning [18] methods—as we clarified earlier, we excluded these techniques in this paper for a more focused and more in-depth comparative study. The reason we summarized the detection technique underlying each of the chosen tools above is because we will refer to those technical details in discussing our empirical results later on.

3.4. Metrics and Measurement

In our study, we considered *accuracy* and *efficiency* (time cost and memory usage) as our metrics. For the accuracy metric, we considered precision, recall and F1 score. We computed the accuracy both for each category and for the entire set of chosen samples in each of the two datasets as a whole. For the efficiency metrics, we considered the CPU time cost and the peak memory usage incurred by the detection process with each detector. We evaluated the scalability of these detectors in terms of time costs and peak memory usage by examining how these efficiency measures change with varying subject sizes. We ran all of our experiments on an HP server of a 10-core Intel Xeon E7 2.4GHz CPU and 512GB RAM, running 64-bit Ubuntu 16.04.

To compute the recall, precision and F1 score, we compared the detection results of each detector with the ground truth associated with each sample. Accordingly, we identified true/false positives/negatives as follows. The results that report vulnerabilities on positive samples were counted as true positives. The results that report no vulnerability on positive samples were counted as false negatives. The results report vulnerabilities on negative samples were counted as false positives. The results that report no vulnerability on negative samples were counted as true negatives.

Some of the detectors could not finish the detection run against certain sample programs within a reasonable amount of time. Therefore, we had to set a detection time threshold, such that any detection run that takes longer than the threshold is killed and counted as a *timeout* case. We set this threshold to 24 hours (1440 minutes) in an attempt to minimize the number of timeout cases, since no valid accuracy or efficiency results can be collected from these cases. Accordingly, the accuracy results we report are those computed based on non-timeout cases.

We computed the statistical significance of F1 score

difference between each pair of detectors and the magnitude of the difference in terms of effect size. We chose Wilcoxon signed-rank tests [42] to compute the p value at the 0.95 confidence level, thus the differences with $p \leq 0.05$ would be considered significant. We used Cliff's Delta [12] as the effect-size measure and interpret the strength of this measure as per the commonly adopted breakdown [34]: Given a Cliff's Delta value d , 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 $|d| > 0.474$.

4. Results

We present our study results and major findings as per the five research questions. For each of the first three questions, we do so for the two benchmark datasets separately. We then looked across these datasets for RQ4 and RQ5.

4.1. RQ1: Accuracy

We first look at the accuracy (in terms of precision, recall, and F1 score) of the five detectors as a whole (i.e., as representatives of state-of-the-art code-analysis-based memory error vulnerability detectors) to understand where current vulnerability defense techniques are. Then, we provide insights and recommendations based on our results.

4.1.1. Software-Analysis-Benchmark

Figures 2, 3, 4 shows the recall, precision, and F1 score of each of the five detectors (each group) against each of the 14 vulnerability categories (each bar). Each category contains equal numbers of positive and negative samples. In each of the figures, each group of bars shows the respective accuracy metric of one detector against these categories. Each bar in a group indicates the metric of that detector against one category and is distinguished from others by the fill patterns as listed at the bottom of the figure. The last bar of each group in the figures shows the metric of the detector against the entire dataset.

The results indicated that the detectors had in many cases perfect precision, except for ADDRESSSANITIZER and CBMC against the stack underrun samples, which saw false positives. Note that all the vulnerabilities reported by purely dynamic analysis based tools were true positives. The reason was that the insertion of assertion statements in CBMC is based on a static analysis hence conservative, so is the insertion of memory status check logic during the static instrumentation in ADDRESSSANITIZER. While ensuring soundness, the conservativeness inherently brings about false positives. To mitigate this issue, more precise static analysis could be adopted in these tools, which however would come at the cost of risking soundness and/or incurring higher overhead.

ADDRESSSANITIZER and CBMC both reported false positives for stack underrun vulnerabilities due to the conservative nature of the static analysis they use. In contrast, other vulnerability detectors reported no false positives.

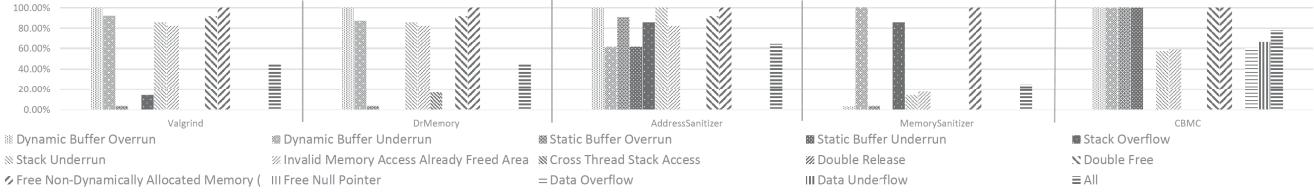


Figure 2: Recall of the five compared memory error vulnerability detectors, per category and overall, for *Software-Analysis-Benchmark*.

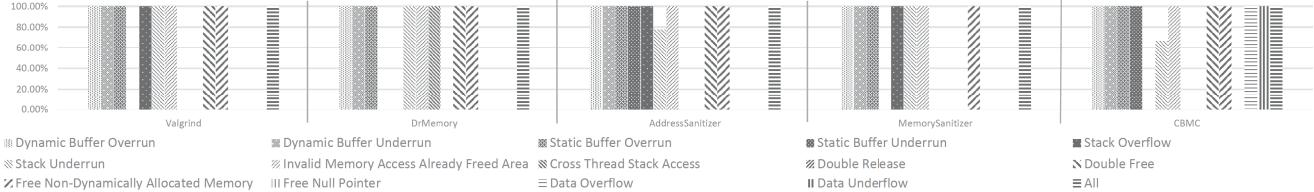


Figure 3: Precision of the five compared memory error vulnerability detectors, per category and overall, for *Software-Analysis-Benchmark*.

Generally, the recall was low with these detectors for many vulnerability categories. CBMC had relatively the highest recall, because of the completeness advantage of static analysis against simple programs (less than 300 depth of iterations or recursions), where all the control flow paths and assertions were traversed by CBMC. With the perfect precision in general, this led to CBMC having the highest F1 score as well. In contrast, dynamic analysis did not seem to bring accuracy benefits compared to static analysis for vulnerability detection here. The reason was that the dynamic analysis was commonly limited to the specific executions—it did not execute all program paths hence missed some vulnerabilities.

For this reason, our evaluation and benchmark datasets may not be fair, especially for the dynamic detectors, since we only considered limited run-time inputs for the samples; using two disparate datasets was attempted to reduce, but not eliminate, this limitation. A typical way to mitigate this general challenge to dynamic analysis due to the limited coverage of limited run-time inputs considered is to run each program multiple times with different inputs while running the detector against each execution. This of course would incur higher overall costs of the detection. Another promising solution would be to employ fuzzing to automatically generate more run-time inputs to more extensively explore the run-time state space of the samples, so as to capture vulnerabilities more comprehensively.

The dynamic detectors had lower recall than the static detector CBMC, because dynamic analysis was commonly limited to the specific executions. This challenge to the dynamic detectors may be overcome by considering multiple executions with different existing run-time inputs or more extensive inputs automatically generated through fuzzing.

Overall, the accuracy of these detectors varied widely: 0% to 100% precision, 0% to 100% recall, and 0% to 100% F1 score per category, indicating most of the techniques worked extremely well on certain kinds of vulnerabilities yet quite poorly on others. The main reason for this large variation was that each of the underlying detection techniques tended to focus on particular kinds of vulnerabilities, leaving others largely unattended.

For instance, **MEMORYSANITIZER** is designed to fast detect use of uninitialized memory. Thus, arithmetic overflows, corresponding to data overflow and data underflow vulnerabilities in the dataset, are not within its detection scope. Likewise, **VALGRIND** is a dynamic detector by nature, not capable of detecting source-code level buffer errors (corresponding to static buffer overrun and static buffer underrun vulnerabilities in the dataset). In all, none of these tools were able to detect a vast variety of vulnerabilities at the same time. The last bar of each group shows the overall accuracy, computed by treating the entire benchmark dataset as a whole: The two detectors

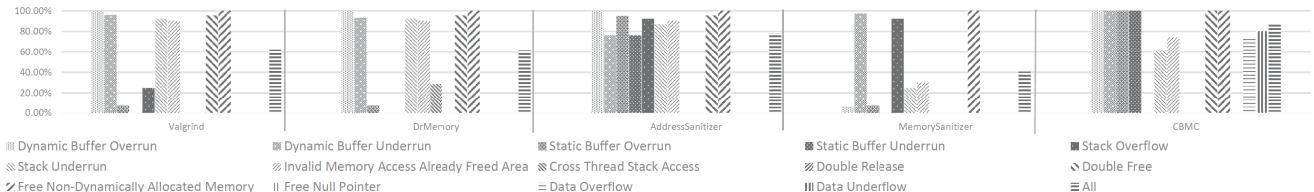


Figure 4: F1 score of the five compared memory error vulnerability detectors, per category and overall, for *Software-Analysis-Benchmark*.

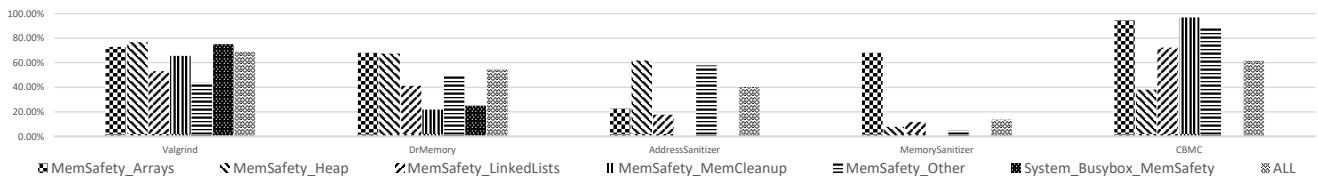


Figure 5: Recall of the five compared memory error vulnerability detectors, per category and overall, for *SV-Benchmark*.



Figure 6: Precision of the five compared memory error vulnerability detectors, per category and overall, for *SV-Benchmark*.

with static analysis, CBMC and ADDRESSSANITIZER, achieved the highest F1 score (87.3% and 77.7%, respectively). VALGRIND was the best-performing purely dynamic tool (62.4% F1 score), while MEMORIESANITIZER performed the worst (40.5% F1 score).

The accuracy of these detectors varied widely. Most of them worked very well on certain kinds of vulnerabilities yet poorly on others, given their different focus on different types of vulnerabilities.

4.1.2. SV-Benchmark

Figures 5, 6, and 7 show the recall, precision, and F1 score of the five detectors we studied against the 6 vulnerability categories, in the same format as Figures 2, 3, and 4, respectively. For instance, VALGRIND had nearly 53% recall, 37% precision, and 44% F1 score against the category MemSafety_LinkedLists. For each detector, the result shown in these figures was computed only from samples for which the detector finished the analysis without timeout (as detailed in Section 4.3.2).

We found that VALGRIND, a heavy-weight dynamic detector, achieved the highest overall F1 accuracy (72%), while the lightweight compiler-based tools, ADDRESSSANITIZER and MEMORIESANITIZER, had relatively poor F1 (54% and 24%, respectively) against this benchmark dataset. This advantage was attributed not really to the detection being backed by a dynamic analysis—in fact, MEMORIESANITIZER is a purely dynamic detector as VALGRIND, and ADDRESSSANITIZER also incorporates a dynamic analysis phase. The merit of VALGRIND here is more due to the *disassemble and*

resynthesize (D&R) mechanism and simulated guest CPU adopted in the tool. Thus, the mere fact that a detector is static, dynamic, or hybrid does not necessarily have accuracy implications. What really matters is the specific detection strategy used.

Overall, these detectors had large variations in accuracy across the 6 vulnerability categories. For instance, CBMC achieved 98% F1 score against MemSafety_MemCleanup samples, for which ADDRESSSANITIZER and MEMORIESANITIZER had 0% F1 score. ADDRESSSANITIZER had relatively poor recall (23%) and F1 score (37%) against MemSafety_Arrays but better recall (62%) and F1 score (76%) against MemSafety_Heap. In contrast, MEMORIESANITIZER had higher recall (68%) and good F1 score (81%) against MemSafety_Arrays but poor recall (8%) and F1 score (14%) against MemSafety_Heap. CBMC had relatively poor recall (38%) and F1 score (52%) against MemSafety_Heap but had almost perfect recall and F1 score against MemSafety_Arrays (95% and 97%, respectively) and MemSafety_MemCleanup (95% and 98%, respectively). Treating all the samples in this dataset as a whole, VALGRIND achieved the highest F1 score (72%) while MEMORIESANITIZER had the worst F1 score (24%).

Our results indicate that the studied techniques worked well on certain categories but poorly on others, confirming the finding from the experiments on the other benchmark dataset. The accuracy differences drill down to the differences in the techniques underlying these detectors, which we further discuss via case studies in Section 4.4.

Looking at the five detectors as representatives of current state-of-art memory error vulnerability detection techniques, we found that the highest recall, precision, and



Figure 7: F1 score of the five compared memory error vulnerability detectors, per category and overall, for *SV-Benchmark*.

F1 score was 68%, 99%, and 72%, respectively. For practical use, this level of accuracy may not be sufficient. Generally, most detectors had higher precision than recall for a given set of samples, indicating that among the vulnerabilities that have been detected, there were not many false positives, yet there were still many vulnerabilities that these state-of-the-art detectors did not catch. A main reason is that these detectors chose to prioritize precision over recall in their design [27, 9, 35, 40, 20], which is justifiable with respect to the earlier finding that users (e.g., developers) tended to seek for tools that give fewer false positives hence save defect inspection/confirmation time [11]. Meanwhile, this also suggests a recommendation for further improving the overall accuracy through a better tradeoff between the precision and recall.

For the overall detection accuracy, the detection strategy used by a tool is more important than the type of analysis technique (static, dynamic or hybrid) applied.

4.1.3. Summary

The five detectors had overall high precision on both datasets when the recall was not zero. Yet against some categories the recall was so low that the F1 score was nearly zero. Treating all the categories as a whole in each dataset, the five detectors had moderate F1 score, even though the best result against *SV-Benchmark* was worse than the best result against *Software-Analysis-Benchmark*. These findings suggest not only that the accuracy of these detection tools can be further improved but also that the accuracy measured against different datasets may vary.

4.2. RQ2: Statistical Accuracy Comparison

After looking at the status quo of code-analysis-based memory error vulnerability detection in terms of accuracy, we now look comparatively into the five detectors chosen in this aspect. Tables 3 and 4 show the analysis results for the 10 possible pairs of comparison. For each pair, the two data groups are the per-vulnerability-category F1 score of the two detectors in the pair. For a pair A-B, a positive (negative) effect size indicates that the metric (F1 score) of A is smaller (greater) than that of B. We highlighted in boldface the cases where the *p* value is significant (i.e., ≤ 0.05) and effect size is statistically large (i.e., $|d| > 0.474$).

4.2.1. Software-Analysis-Benchmark

The results of our two statistical analyses are shown in Table 3. The numbers indicate that only CBMC was significantly more accurate than MEMORIESANITIZER with a statistically large difference in terms of the F1 score. This is consistent with the foregoing observation that CBMC had the highest, while MEMORIESANITIZER had the lowest, accuracy according to our results (Figure 4). Between MEMORIESANITIZER and ADDRESSSANITIZER, the accuracy difference was significant but not large (only with a medium effect size of 0.357): the negative sign means the

Table 3

Statistic significance and size of F1 score differences between each pair of the studied detectors, for *Software-Analysis-Benchmark*

Detector Categories	Pair of detectors	<i>p</i> value	effect size
Runtime - Runtime	VALGRIND-DRMEMORY	1	-0.071
Compiler Based - Compiler Based	ADDRESSSANITIZER-MEMORIESANITIZER	0.035	-0.357
	VALGRIND-ADDRESSSANITIZER	0.281	0.071
	DRMEMORY-ADDRESSSANITIZER	0.402	0
	DRMEMORY-MEMORIESANITIZER	0.205	-0.214
Runtime - Static	VALGRIND-MEMORIESANITIZER	0.247	-0.143
	VALGRIND-CBMC	0.236	0.214
	DRMEMORY-CBMC	0.236	0.214
Compiler Based - Static	ADDRESSSANITIZER-CBMC	0.635	0.214
	MEMORIESANITIZER-CBMC	0.032	0.571

Table 4

Statistic significance and size of F1 score differences between each pair of the studied detectors, for *SV-Benchmark*

Detector Categories	Pair of detectors	<i>p</i> value	effect size
Runtime - Runtime	VALGRIND-DRMEMORY	0.259	-0.333
Compiler Based - Compiler Based	ADDRESSSANITIZER-MEMORIESANITIZER	0.251	-0.333
	VALGRIND-ADDRESSSANITIZER	0.004	-0.667
	VALGRIND-MEMORIESANITIZER	0.000	-1
	DRMEMORY-ADDRESSSANITIZER	0.047	-0.667
Runtime - Static	DRMEMORY-MEMORIESANITIZER	0.001	-0.844
	VALGRIND-CBMC	0.400	0.000
	DRMEMORY-CBMC	0.259	0.000
Compiler Based - Static	ADDRESSSANITIZER-CBMC	0.045	0.167
	MEMORIESANITIZER-CBMC	0.011	0.833

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 detection accuracy for this dataset.

The only purely static detector was significantly more accurate than the weakest purely dynamic detector, which corroborated the completeness advantage of static approaches against simple programs.

4.2.2. SV-Benchmark

Like for the other dataset, we performed pairwise comparisons among the five detectors against the *SV-Benchmark* dataset via the same statistical analyses.

Our results revealed that half (five) of the paired comparisons came with statistically *significant and large* accuracy differences. Between the other five pairs, the two detectors either had no significant difference in accuracy, or the differences were at most moderate (medium). Nevertheless, the five pairs with significant and large differences covered all of five detectors we studied, which allowed us to assess the comparative strengths and weaknesses of the entire set of tools.

Both the two heavyweight, purely dynamic analysis based detectors (VALGRIND and DRMEMORY) and the purely static detector (CBMC) were significantly and largely more accurate than the lightweight, compiler-based detectors (ADDRESSSANITIZER and MEMORIESANITIZER). One reason lies in the efficiency optimization mechanism of lightweight compiler-based dynamic detectors, as in ADDRESSSANITIZER and MEMORIESANITIZER. The optimization led to certain losses of completeness of the detection. Given that ADDRESSSANITIZER is a hybrid detector, the contrasts further suggest that combining static

and dynamic analysis did not seem to be necessarily better than purely static or dynamic analysis, at least for memory error vulnerability detection.

While significantly *different* from both lightweight compiler-based detectors (ADDRESSSANITIZER and MEMORIESANITIZER), the accuracy of the purely static detector (CBMC) was significantly *higher* only than MEMORIESANITIZER. One reason is that ADDRESSSANITIZER has a compile-time instrumentation module that performs semantic analysis of source code, while MEMORIESANITIZER is a purely dynamic detector which does not have the semantic check. The static semantic check helped improve the detection accuracy.

For memory error vulnerability detection, while hybrid analysis did not necessarily outperform purely static or dynamic analysis, incorporating a static phase for semantic check helped a dynamic detector achieve higher accuracy.

Meanwhile, between any of the two purely dynamic detectors (VALGRIND and DRMEMORY) and the purely static detector (CBMC), the F1 score difference was not significant. This was mainly because CBMC had a higher recall but lower precision compared to VALGRIND and DRMEMORY, and the higher recall was offset by its lower precision, resulting in overall similar F1 measures. This further confirms the essence of balancing precision and recall in improving the overall accuracy, as discussed earlier.

Static analysis tended not to generally have significantly different F1 accuracy from dynamic analysis for memory-error vulnerability detection, due to the offsetting effects between their intrinsic precision and recall advantages/disadvantages.

4.2.3. Summary

Only one pair of detectors (the best versus the worst) showed a statistically significant and large difference against *Software-Analysis-Benchmark*, while five pairs of detectors showed significant differences against *SV-Benchmark*. In other words, benchmark selection did play a role in telling the performance differences among compared detectors. If two detectors did not appear to have statistically significant and large differences in detection performance against a benchmark dataset, that does not necessarily mean the two detectors are actually close in their detection capabilities. Our results on comparing the accuracy of the five detectors between the two benchmark datasets indicate that these detectors showed greater differences against more complex program samples.

4.3. RQ3: Efficiency

We examine the efficiency (in terms of time costs, memory usage, and scalability) of the five detectors as a whole, while also comparing among these detectors in this regard to understand the efficiency of current vulnerability

defense techniques and to provide actionable insights and recommendations based on the empirical results.

4.3.1. Software-Analysis-Benchmark

Time Costs. Table 5 lists the per-sample average time costs incurred by each of the studied detectors, for each vulnerability category (third to sixteenth rows). 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 program sample if necessary). To help understand the run-time (slowdown) overheads incurred by *run-time tools*, the table (second and eighth columns) also lists the average original execution time per sample (*Directly*). Given the negligible variance of these efficiency numbers in any vulnerability category—as justified by the closeness in size and complexity among program samples in this dataset, we only report the means here and do not show the full distribution as we do for the efficiency results against the other benchmark dataset.

Our results show that the costs of vulnerability detection with these detectors were hundreds or even thousands of times greater than the *Directly* costs. This is because the detectors have common steps like module initialization and code analysis and transformation, which incurred non-trivial amount of time, while the costs of running the simple and short programs in this dataset were generally tiny.

In contrast to the other tools, VALGRIND had the highest time costs, because it has the most sources of costs (e.g., initializing its heavy-weight core and the *Memcheck* module, translating code to IR, dynamic code instrumentation and optimization, running code in its simulated CPU) each incurring substantial overhead. Similar to VALGRIND by nature, DRMEMORY incurred the second highest time costs in an average case. Yet the costs were only about half of those incurred by VALGRIND mainly because it ran programs on a real CPU, avoiding the overhead of CPU virtualization. With the two compiler-based detectors, ADDRESSSANITIZER and MEMORIESANITIZER, just compiling and instrumenting these very small programs incurred much less cost.

Compared to these dynamic tools, CBMC incurred relatively moderate costs but with larger variations. This is because it analyzes control flow to determine where to place the assertions against memory access validity, followed by solving CNFs for paths reaching those assertions (Section 3.3). These steps intuitively cost higher wherever the control flow is more complex (e.g., having more or deeper nested branching structures). As a result, it saw the maximal costs (about 21 seconds, as highlighted in boldface) with the positive stack underrun samples and negative stack overflow samples, which do contain more complex control flow than others in this dataset.

Table 5

Time costs (ms) of the compared detectors, per category (size in parentheses) and overall, against the *Software-Analysis-Benchmark* dataset

Vulnerability Category	Positive Samples						Negative Samples					
	Directly	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC	Directly	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC
Dynamic Buffer Overrun (64)	<0.1	1,018.8	578.8	26.8	26.8	290.4	<0.1	1,058.4	592.2	29.8	26.8	298.8
Dynamic Buffer Underrun (78)	<0.1	1,019.2	575.0	25.6	30.2	362.6	0.2	1,052.0	577.2	29.4	27.2	343.8
Static Buffer Overrun (108)	<0.1	1,020.6	568.7	27.1	27.4	214.7	<0.1	1,021.8	591.9	30.7	27.6	208.9
Static Buffer Underrun (26)	2.5	934.1	506.1	29.8	27.8	139.4	<0.1	1,019.4	626.1	30.8	27.2	139.7
Stack Overflow (14)	<0.1	993.7	573.7	33.1	28.4	139.4	<0.1	1,022.3	573.1	33.1	26.7	20,875.4
Stack Underrun (14)	0.6	1,022.9	4,948.5	26.9	30.7	21,775.4	<0.1	1,029.7	669.7	28.0	28.4	1,187.4
Invalid Memory Access to Already Freed Area (34)	0.2	1,023.1	572.0	26.8	28.1	366.6	<0.1	1,084.9	575.1	32.0	27.8	350.1
Cross Thread Stack Access (12)	<0.1	1128.6	614.7	32.0	30.3	250.0	<0.1	1132.0	652.7	32.0	29.0	244.7
Double Release (12)	<0.1	1,085.3	630.7	35.3	26.0	229.3	1.3	1,084.7	643.3	32.0	30.3	229.3
Double Free (24)	0.3	1,007.7	571.3	28.3	26.6	166.0	0.3	1,017.3	575.0	32.7	28.3	164.7
Free Non-Dynamically Allocated Memory (32)	<0.1	1,071.5	716.75	31.0	30.7	183.5	<0.1	1,014.7	615.0	32.0	27.0	183.5
Free Null Pointer (28)	<0.1	1,016.3	550.3	25.7	27.6	321.1	<0.1	1,105.7	573.1	28.3	29.3	456.0
Data Overflow (50)	<0.1	1,020.6	627.5	28.9	27.6	133.8	<0.1	1,016.2	618.6	29.1	27.6	137.6
Data Underflow (24)	0.3	1,013.0	567.3	31.3	29.0	130.0	0.3	1,122.7	592.3	31.3	29.0	132.0
All (520)	<0.1	1,021.3	702.5	28.2	28.1	822.4	<0.1	1,047.6	597.8	30.4	28.2	829.8

Table 6

Peak memory usage (MB) of the compared detectors, per category (size in parentheses) and overall, against the *Software-Analysis-Benchmark* dataset

Vulnerability Category	Positive Samples						Negative Samples					
	Directly	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC	Directly	Valgrind	DrMemory	Address Sanitizer	Memory Sanitizer	CBMC
Dynamic Buffer Overrun (64)	2.9	9.0	6.8	6.0	5.1	8.8	2.8	8.1	6.2	5.5	4.9	8.7
Dynamic Buffer Underrun (78)	2.8	9.0	6.9	5.9	5.2	11.2	2.8	8.4	6.1	5.7	4.8	10.8
Static Buffer Overrun (108)	3.0	8.9	6.6	6.0	5.3	4.5	2.9	8.0	6.6	5.6	4.9	4.1
Static Buffer Underrun (26)	3.5	8.5	6.3	5.7	5.0	5.0	3.3	8.3	6.7	5.5	4.9	4.3
Stack Overflow (14)	3.2	8.6	6.4	5.8	4.8	5.3	3.3	8.7	6.5	5.6	4.6	69.8
Stack Underrun (14)	3.6	8.8	6.5	5.9	4.9	68.7	3.4	8.5	6.6	5.8	4.6	5.3
Invalid Memory Access to Already Freed Area (34)	3.1	8.7	6.7	5.7	5.2	8.8	3.1	8.2	6.5	5.8	4.7	8.6
Cross Thread Stack Access (12)	3.1	8.9	6.6	6.1	5.1	7.2	3.1	8.4	6.3	5.9	4.9	7.4
Double Release (12)	2.8	8.6	6.8	6.2	5.0	7.6	3.0	8.1	6.1	5.5	4.8	7.3
Double Free (24)	2.8	8.7	6.7	5.8	5.4	8.4	2.8	8.2	6.1	5.5	5.0	8.1
Free Non-Dynamically Allocated Memory (32)	2.9	8.7	6.6	5.9	5.3	5.8	2.8	8.2	6.2	5.6	5.1	5.3
Free Null Pointer (28)	2.8	8.7	6.5	6.0	5.1	7.8	2.7	8.4	6.5	5.9	4.7	7.2
Data Overflow (50)	2.6	8.6	6.6	6.0	4.9	4.8	2.5	8.2	6.4	5.5	4.6	4.0
Data Underflow (24)	2.5	8.5	6.6	5.8	4.8	4.7	2.5	8.1	6.4	5.7	4.6	4.0
All (520)	2.9	8.8	6.7	5.9	5.1	8.7	2.9	8.2	6.4	5.6	4.8	8.3

Overall, the two purely dynamic detectors incurred higher overhead than the other three across all vulnerability categories. The only static detector saw higher overhead with more complex control flow.

In all, these detectors were generally extremely fast against the samples used in terms of the (small) absolute cost measures—taking about one second per sample in most cases. Yet the overheads incurred by dynamic tools were substantial in terms of run-time slowdown. Also, there were no significant differences in efficiency between positive and negative samples for any of these detectors. This indicated that the time efficiency was not affected much by whether the programs had vulnerabilities or not.

The detectors were generally fast in absolute terms, mostly taking just about one second per sample. For any detector, the time cost was not much affected by whether or not the sample is vulnerable.

Memory Usage. Table 6 lists the per-sample average peak memory usage incurred by each of the studied detectors, for

each vulnerability category. Similarly to how the time costs are presented, we also show the peak memory usage for positive and negative samples separately, and list the corresponding memory usage of running the original sample directly as a contrast to show the memory overhead of vulnerability detection with these tools.

The numbers show that, in absolute terms, these detectors mostly used very little memory (less than 10 MB) against each of the samples in *Software-Analysis-Benchmark*. In terms of the memory overhead, the four dynamic tools (VALGRIND, DRMEMORY, ADDRESSSANITIZER, and MEMORIESANITIZER) consumed 1.5-3x of the memory consumed by running the samples directly. Specifically, VALGRIND used the most memory because of its bit-to-bit shadow memory mapping, more than DRMEMORY and ADDRESSSANITIZER which adopted 1-byte-to-2-bit and 8-byte-to-1-byte shadow memory mapping, respectively. MEMORIESANITIZER also used bit-to-bit shadow memory mapping, but it only mapped user-defined memory. Thus, it used less memory than VALGRIND. The variations in

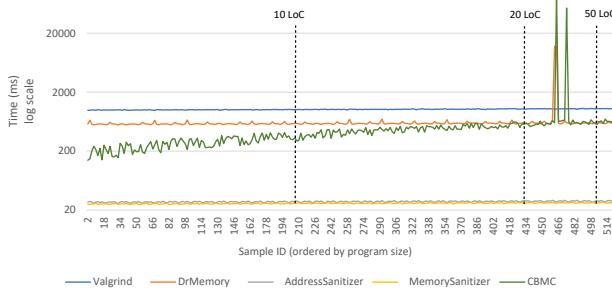


Figure 8: Scalability of the compared detectors against individual samples in the *Software-Analysis-Benchmark* dataset in terms of time costs (y axis).

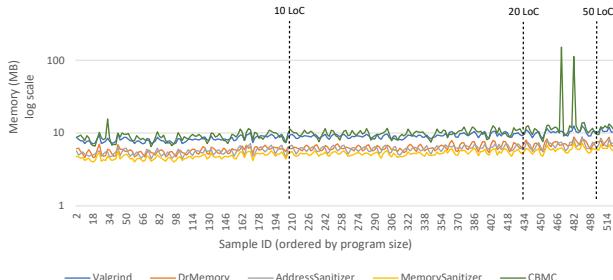


Figure 9: Scalability of the compared detectors against individual samples in the *Software-Analysis-Benchmark* dataset in terms of peak memory usage (y axis).

memory usage across the vulnerability categories and the two outlier cases (highlighted in boldface) as seen by CBMC can be explained by the same reasons that explain these variations and outliers in the time costs (Table 5).

We also found no significant difference in the memory usage of any of these detectors between negative and positive samples within any vulnerability category.

The detectors generally used very little memory (less than 10 MB) for each sample. The differences in memory usage among the detectors are justified by their different shadow memory mapping granularity.

Scalability. Figures 8 and 9 show the scalability of the studied detectors in terms of time costs and memory usage (y axis, in log scale), respectively, against individual samples (sorted by code sizes non-descendingly as listed on the x axis) in the *Software-Analysis-Benchmark* dataset. Overall, the results show that, despite a few outliers, these detectors had very good scalability against this dataset—as the program sizes increased, the time costs and peak memory usage kept almost constant, except for CBMC had a linear increase in time costs yet the slope was very small.

Among the five detectors, we found that the scalability of the four dynamic detectors (VALGRIND, DRMEMORY, ADDRESSSANITIZER, and MEMORIESANITIZER) was slightly better than the static detector CBMC. This is evidenced by the observation that the dynamic detectors had even smaller efficiency variations across samples of different sizes than the static detector, and that the outliers were mostly seen by the static detector also. This can be explained by the sensitivity of CBMC to the characteristics

of program structures and code constructs (e.g., branches, control flow complexity) that do vary with code sizes, with respect to the internals of this detector.

All the five detectors showed generally high scalability against the samples in the *Software-Analysis-Benchmark* dataset. The four dynamic detectors appeared to be even more scalable than the static detector CBMC due to the nature of the detection technique in the latter.

4.3.2. SV-Benchmark

Time Costs. Figure 10 shows the violin plots of the time costs for the five detectors against the 6 vulnerability categories in the *SV-Benchmark* dataset. In each of the subfigures, the first violin plot shows the distribution of the time costs of running the samples directly, as a baseline (noted as *Directly*), and each of the other four violin plots associated with a detector shows the distribution of the time costs of that detector against the vulnerability category associated with the subfigure. For example, against MemSafety_MemCleanup, the median of the time costs with VALGRIND was 11 minutes and the maximum was 20 minutes—the outlier costs were up to over 50 minutes.

The time costs of the five detectors against the 6 categories varied widely. For instance, for MemSafety_Arrays and MemSafety_Heap samples, CBMC spent over 200 minutes in the worst case (for one sample) while MEMORIESANITIZER finished checking any sample in no more than 1 minute. Against MemSafety_MemCleanup samples, VALGRIND and DRMEMORY spent relatively longer time in both average (4 and 3 minutes, respectively) and maximum (20 and 13 minutes, respectively) cases than other detectors. The results revealed that some detectors had noticeably higher efficiency against certain vulnerability categories than others.

Specifically, when compared to other detectors, CBMC took more time on the MemSafety_Arrays, MemSafety_Heap, MemSafety_LinkedList, and Systems_BusyBox_MemSafety categories, but noticeably less on the other two categories. This was because the key steps for vulnerability detection in CBMC, inserting assertions and solving constraints, were expensive for complex control flows (e.g., due to the presence of more branching structures), which were common in these four categories. Among the four dynamic detectors, VALGRIND spent the most time against any vulnerability category, while DRMEMORY and ADDRESSSANITIZER spent less and MEMORIESANITIZER spent the least. The reason was that, VALGRIND ran the programs on its *guest* (virtual) CPU, which significantly impacted the efficiency, while the others ran the programs on the real CPU. MEMORIESANITIZER was faster than both DRMEMORY and ADDRESSSANITIZER mainly because MEMORIESANITIZER is dedicated to checking only against uninitialized memory while the other two aim to check a much broader range of memory error vulnerabilities hence need more extensive/complex instrumentations and

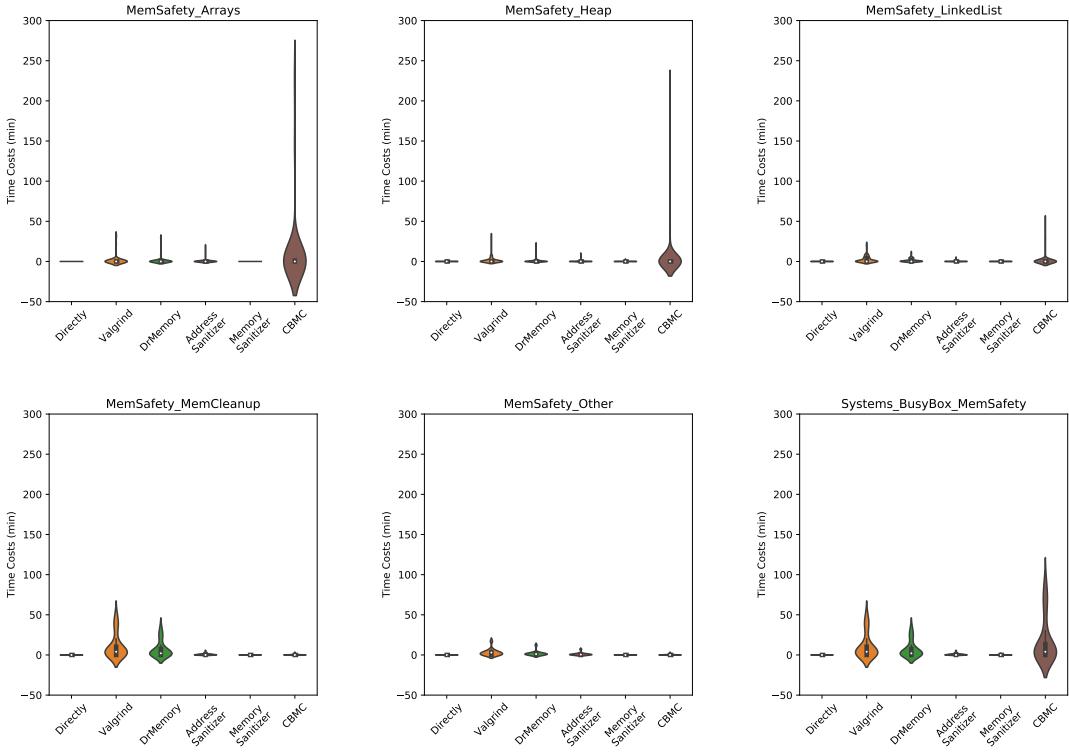


Figure 10: Distribution of the time costs (y axis, in minute) of the five compared memory error vulnerability detectors against individual samples in the *SV-Benchmark* dataset, shown separately per vulnerability category (as shown as the chart title).

dynamic analysis. Lastly, ADDRESSSANITIZER was still faster than DRMEMORY because the former statically instruments the programs, inherently more efficient than dynamic instrumentation with DRMEMORY.

The efficiency of the static detector CBMC appeared to be more sensitive to control flow complexity, due to its vulnerability detection strategy relying on control flow analysis (for assertion insertions).

As for RQ1, the efficiency results discussed above did not include timeout cases. We list the numbers of timeout cases in Table 7 per detector and vulnerability category—there were no such cases in the *Software-Analysis-Benchmark* dataset. We found that the numbers of such cases were particularly high against MemSafety_LinkedLists and MemSafety_Other samples for any of the detectors. For instance, the number was 13 for CBMC and 14 for other detectors against MemSafety_LinkedLists samples, while the numbers were greater than 10 against MemSafety_Other samples. Against any other categories, the number was less than 5.

To understand the causes, we conducted manual inspections and found that there were some samples in the MemSafety_LinkedLists and MemSafety_Other categories dealing with very deep (over 10,000 in some samples) or

Table 7
Number of timeout cases

Vulnerability Category (Size)	VALGRIND	DRMEMORY	Address Sanitizer	Memory Sanitizer	CBMC
MemSafety_Arrays (71)	0	0	0	0	3
MemSafety_Heap (180)	1	1	1	1	4
MemSafety_LinkedLists (103)	14	14	14	14	13
MemSafety_MemCleanup (32)	0	0	0	0	0
MemSafety_Other (48)	13	13	10	10	12
System_Busybox_MemSafety (40)	1	1	1	1	0
All (474)	29	29	26	26	32

even unbounded nested iterations or recursions, which led to particularly high resource consumption for both static and dynamic detection hence the timeouts. The dominating timeout cases in the two categories were largely attributed to those samples, which we confirmed were all real-world applications. Note that compared to the other categories, MemSafety_Other is a mixed category (which is noted by the name of this category)—the other (i.e., non-timeout) samples caused very little time (see Figure 10).

Memory Usage. Figure 11 depicts the distributions of memory usage by the five detectors in the same format as Figure 10, except for that the y axes show the peak memory usage instead of time costs.

The results show that the peak memory usage of these detectors was generally moderate and in a reasonably small

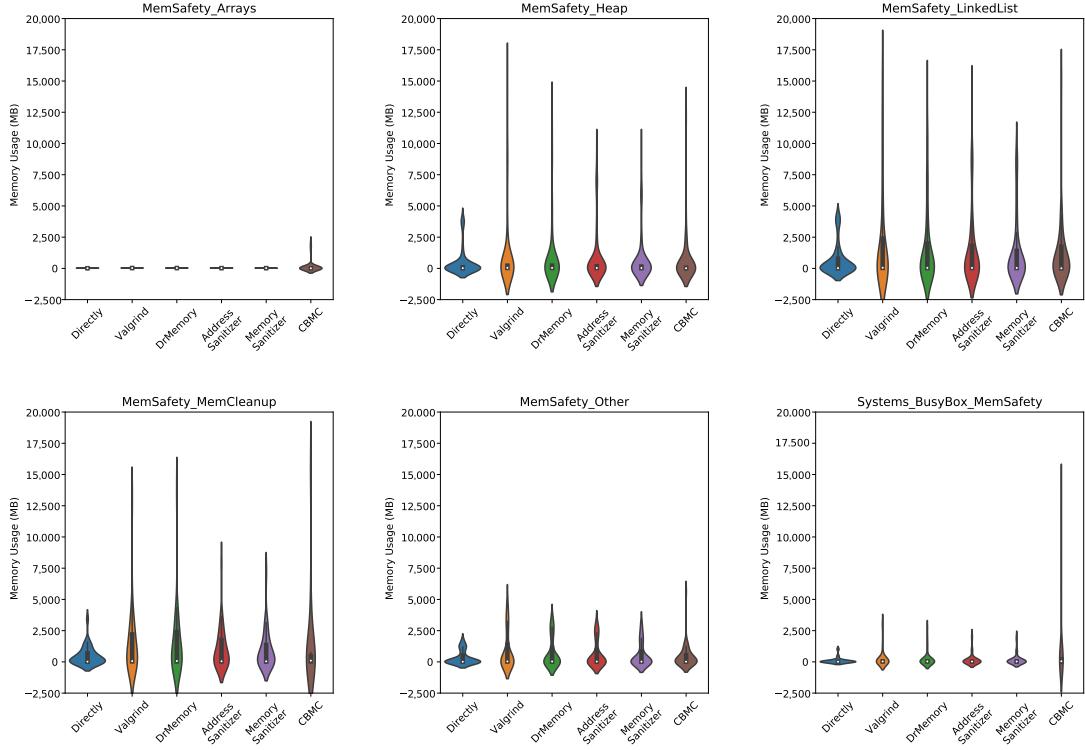


Figure 11: Distribution of the peak memory usage (y axis, in MB) of the five compared memory error vulnerability detectors against individual samples in the *SV-Benchmark* dataset, shown separately per vulnerability category (as shown as the chart title).

range—checking most of the samples needed no more than 2.5GB memory, despite a few outliers (taking up to 16GB memory). On the other hand, comparing to the memory usage of running the original samples directly revealed that the memory overhead of these detectors was generally moderate too (below 1.5x)—albeit quite high (about 4x) with the outlier cases.

The peak memory usage of these detectors was generally moderate and in a reasonably small range, despite a few outlier cases. The differences in memory usage among the detectors were justified by their different levels of shadow memory granularity.

Scalability. Figures 12 and 13 show the scalability of the five detectors against individual samples in *SV-Benchmark* in terms of time costs and peak memory usage, in the same format as Figures 8 and 9, respectively.

The results show that neither the time cost nor peak memory consumption increased continuously as the sample code size grows. Meanwhile, the time cost and memory usage were reasonably practical for the majority of the samples, and the peaks were in small numbers. These observations suggest that the detectors generally scaled well to the samples in this dataset. We also noticed that the scalability of all of these detectors, especially the two heavy-weight dynamic detectors VALGRIND and

DRMEMORY, are generally quite sublinear, suggesting opportunities for improving precision and recall at the cost of more computing resources.

All five detectors scaled well to the *SV-Benchmark* dataset, with the heavy-weight dynamic detectors providing the best scalability.

4.3.3. Summary

Our results revealed that all the samples in the *Software-Analysis-Benchmark* dataset cost little time to be checked because of their simple code and data structures. Yet there were more than 25 samples in the *SV-Benchmark* dataset costing more than 24 hours to be checked by any of the detectors. The average time cost per sample in the *SV-Benchmark* dataset was also higher than the average time cost per sample in the *Software-Analysis-Benchmark* dataset. These contrasts indicated that the data and code structure complexity had considerable impact on the efficiency of these detectors.

Contextualizing our efficiency results, we found that earlier efficiency evaluations of the four dynamic detectors [27, 9, 35, 40] against the SPEC 2006 benchmark [17] had results consistent with ours in terms of the relative ranking of these tools in efficiency. Generally, the common conclusion is that VALGRIND is a

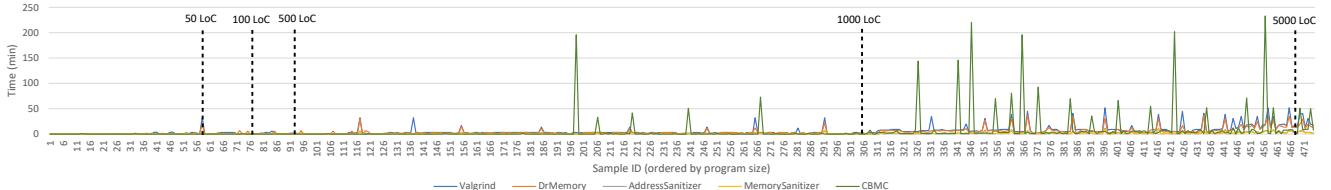


Figure 12: Scalability of the time costs (y axis) of the five compared memory error detectors against individual samples in the *SV-Benchmark* dataset

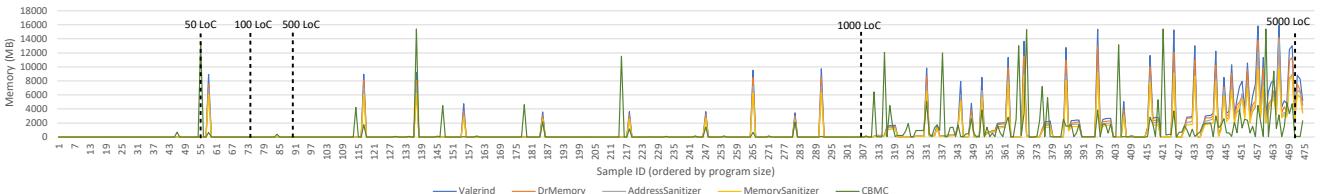


Figure 13: Scalability of the memory usage (y axis) of the five compared memory error detectors against individual samples in the *SV-Benchmark* dataset

heavy-weight framework while ADDRESSSANITIZER and MEMORIESANITIZER are faster memory checkers by design.

4.4. RQ4: Failure Cause Analyses (Case Studies)

We extracted 9 samples in the two benchmark datasets as example cases. For each case, we show how well some of the detectors performed against it, and discuss the reasons behind the successes and failures. Then, we provide actionable insights and/or recommendations based on the results and discussions.

```

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: In-depth inspection: Case 1.

Case 1. Listing 1 shows the excerpt of two static buffer overrun samples. Both VALGRIND and DRMEMORY detected the vulnerability in the second sample but not in the first, although the first seems easier to detect.

The reason was because both tools are purely dynamic detectors, thus they did not check the static buffer size in the source code. Instead, they only checked whether the address (in stack or heap) being accessed was valid. In the first sample, `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.

This suggested that it would have merits for dynamic detectors to analyze the source code if possible. We noticed that the two purely dynamic detectors were able to report

the vulnerable lines when we enabled debugging while compiling the programs. If the dynamic detectors were able to utilize the debugging information in the executable programs, this kind of false negatives could be reduced.

VALGRIND and DRMEMORY failed to detect static buffer overruns induced by statically set buffer sizes, because as purely dynamic detectors they did not check such sizes. This issue can be mitigated by utilizing the debugging information in the executable programs when available.

```

1 void first()
2 {
3     char buf[5];
4     buf[5] = 1;      /*Tool should detect this line as error*/
5     /*ERROR: buffer overrun */
6     ...
7 }
8 void second()
9 {
10    int buf[5][6];
11    buf[5][5] = 1;   /*Tool should detect this line as error*/
12    /*ERROR: buffer overrun */
13    ...
14 }
15 void third()
16 {
17    int buf[5][6][7];
18    buf[5][5][6] = 1; /*Tool should detect this line as error*/
19    /*ERROR: buffer overrun */
20    ...
21 }
```

Listing 2: In-depth inspection: Case 2.

Case 2. Listing 2 shows three static buffer overrun samples. ADDRESSSANITIZER detected the vulnerability in the first two samples but did not in the third.

The reason was that the instrumentation module of this detector, which is used to check the shadow state for each memory access to the stack or global variables, did not check overrun errors for arrays of 3 or higher dimensions. Instead, as a lightweight detector aiming at high efficiency, ADDRESSSANITIZER only checked arrays of up to 2 dimensions, since arrays of deep dimensions are not as common in real-world software systems.

This indicated that hybrid analysis, as is this detector, did not necessarily outperform purely static or dynamic analysis in detecting memory error vulnerabilities when dealing with complex data structures—besides high-dimension arrays, another example is a deeply embedded linked list as seen in a similar case. A more thorough run-time checking could overcome this challenge.

ADDRESSSANITIZER, a lightweight hybrid detector, did not detect static buffer overrun vulnerabilities in arrays of 3 or higher dimensions, which could be overcome by a more thorough run-time checking.

```

1 void stackoverflow()
2 {
3     double buf[1048576]; /*Tool should detect this line as error*/
4     /*ERROR:Stack overflow*/
5     buf[0] = 1.0;
6     ...
7 }
```

Listing 3: In-depth inspection: Case 3.

Case 3. Listing 3 shows a stack overflow sample, for which CBMC did not detect the stack overflow vulnerability.

The reason was that this type of vulnerabilities was closely related to the maximum stack size dynamically configured at the operating system level, which cannot be determined purely statically by CBMC. All the other (dynamic) detectors were able to detect stack overflows that are associated with dynamically configured run-time environment parameters as they can readily check such run-time configurations.

For a static detector like CBMC to overcome this challenge, one way would be to retrieve relevant environment configurations and make conservative assumptions about possible bounds of the configurations (e.g., maximal stack size possible) in the detection algorithm (e.g., during the assertion generation step of CBMC). Alternatively, such bounds can be explicitly specified by users.

CBMC as a static detector did not capture stack overflow vulnerabilities due to the dynamically set maximum stack size, which may be overcome by carefully checking such environment parameters.

```

1 void printEven(int i){__VERIFIER_assert((i%2)==0);}
2 void printOdd(int i){__VERIFIER_assert((i%2)!=0);}
3 int main()
4 {
5     int array[100000];
6     int i;
7     int num=100001;
8     for(i=0;i<num;i++)
9         if(array[i]%2==0)printEven(array[i]);//Invalid dereference
10    for(i=0;i<num;i++)
11        if(array[i]%2!=0)printOdd(array[i]); //Invalid dereference
12    return 0;
13 }
```

Listing 4: In-depth inspection: Case 4.

Case 4. Listing 4 shows part of a positive sample with array related memory safety violations. All of the five detectors but CBMC identified these vulnerabilities (at Lines 9 and 11).

This failure with CBMC was caused by the large depth of the loop (100,001), greater than the deepest iteration this detector unrolled the loop up to, which was set to 300 due to time and memory cost concerns. It did not check up to the iteration where the vulnerabilities occurred hence failed to detect them. In contrast, the other detectors contained a dynamic analysis that checked all the iterations in this case, thus they were all able to capture the vulnerabilities. For a similar reason, CBMC would miss vulnerabilities in deep recursions too, where a dynamic detector would have a better chance to succeed.

A static detector like CBMC could mitigate such limitations by increasing the loop unwinding depth, at the cost of greater time and memory overhead. While the actual loop bounds are hard to predict statically, a limit that accommodates most situations as per empirical evidence may still work reasonably well in practice. Another way to overcome the challenge would be to precede the static vulnerability detection algorithm with a constant propagation to derive loop bounds at compile time if possible—in the sample here, this would work well.

CBMC failed to detect vulnerabilities in deep loop iterations or recursions due to the limited depth the static analysis was set to unwind up to. On-demand setting of this limit and/or constant propagation may help overcome this challenge to purely static vulnerability detectors like CBMC.

```

1 int main()
2 {
3     int *myPointerA=((void*) 0);
4     int *myPointerB=((void*) 0);
5     {
6         int myNumberA=7;
7         myPointerA=&myNumberA;
8         // scope of myNumberA ends here
9     }
10    int myNumberB=3;
11    myPointerB=&myNumberB;
12    //Invalid dereference for myPointerA
13    int sumOfMyNumbers=*myPointerA+*myPointerB;
14    printf("%d",sumOfMyNumbers);
15    return 0;
16 }
```

Listing 5: In-depth inspection: Case 5.

Case 5. Listing 5 shows part of a positive sample where the dereference of the pointer `myPointerA` is invalid at Line 13 because its valid definition (Line 7) is out of the scope of the dereference. While this vulnerability was correctly detected by ADDRESSSANITIZER and CBMC, it was not by other detectors.

We found that the failure with these other detectors occurred because they tracked all definitions, initializations, assignments, and (de)allocations via shadow memory, thereby accesses to deallocated memory could be detected. However, they did not check source code nor were aware of the language semantics that the variable

address (of `myNumberA`) becomes invalid beyond Line 9—in other languages (e.g., Shell) the address would not become invalid. In contrast, CBMC captured this in its semantic check phase which examines the code in a semantics-aware manner, and ADDRESSSANITIZER detected the invalid access during its compilation process in which the analysis is aware of the language semantics as well.

Apparently, awareness of language semantics would be an effective mitigation of this limitation to dynamic detectors—adding a semantic check at compile time could have helped the other detectors succeed in this case.

Purely dynamic detectors (VALGRIND, DRMEMORY and MEMORIESANITIZER) did not capture out-of-scope dereference vulnerabilities due to their ignorance of language semantics. Incorporating a compile-time semantic check would help mitigate this challenge to purely dynamic detectors.

```

1 //Original sample
2 void entry_point(void){
3     int *p=(int *)ldv_malloc(sizeof(int)); //ldv_malloc = malloc
4 }
5 void main(void){entry_point();}
6
7 //=====
8 //Modified sample
9 void entry_point(void){
10    int *p=(int *)ldv_malloc(sizeof(int));
11    int *q=(int *)ldv_malloc(sizeof(int));
12 }
13 void main(void){entry_point();}

```

Listing 6: In-depth inspection: Case 6.

Case 6. Listing 6 shows a positive sample (Lines 1-5), where a memory-leak vulnerability occurred when the memory allocated at Line 3 did not get freed afterwards. All detectors but DRMEMORY captured this vulnerability.

We found that the failure with DRMEMORY was due to its special detection strategy: It only checks memory leaks in the middle (as opposed to the boundaries) of the heap [9], in order to avoid false positives with C++ programs that allocate memory via the `new[]` operator and `std::string` objects, or with pointers pointing to an instance of a class with multiple inheritance. In both kinds of situations, extra heap memory is justifiably reserved to accommodate the particular memory model of C++. To verify this explanation, we modified the original sample source code (Lines 1-5) and created a modified version, which is shown in Lines 9-13. This corroborated that DRMEMORY indeed only checks memory in the middle of the heap, causing its failure with the original sample here.

This failure was essentially caused by the soundness-precision tradeoff adopted by DRMEMORY. A mitigating strategy would be to bring users in the loop of decision making regarding the tradeoff, by offering options to users that determine whether particular language features should or should not be conservatively considered.

DRMEMORY adopted a special detection strategy that helps it reduce false positives with particular (C++) language features but caused false negatives. An user option for or against such conservative strategies would help deal with this issue.

```

1 static int ldv_m88ts2022_re_reg(
2     struct ldv_m88ts2022_priv *priv,
3     char reg, char *val){
4     ...
5     memcpy(val,buf,1);
6     ...
7 }
8 //A function that will be called when running
9 int alloc_fix_12(struct ldv_i2c_client *client){
10    unsigned char chip_id;
11    //char chip_id;
12    //change "unsigned char" to "char", false positive disappears
13    int ret;
14    ...
15    ret=ldv_m88ts2022_re_reg(priv,0x00,&chip_id);
16    ...
17    //chip_id is initialized, but the tools still report errors
18    switch(chip_id){
19    ...
20 }
21 }
```

Listing 7: In-depth inspection: Case 7.

Case 7. Listing 7 shows part of a *negative* sample. Three of the dynamic detectors, VALGRIND, MEMORIESANITIZER and DRMEMORY, falsely reported the use of uninitialized variable (`chip_id`) at Line 18 as a vulnerability. In fact, this variable is initialized at Line 15 through the function call there. We tried changing the data type from `unsigned char` to `char` at Line 10, then this false positive disappeared with any of these detectors.

The underlying cause had to do with type casting. At Line 3, the function `ldv_m88ts2022_re_reg` accepted a pointer type `char*` for its third parameter while the type of the argument `&chip_id` provided at Line 15 was `unsigned char*`. The three dynamic detectors instrumented the program so as to update the status of `chip_id` in the shadow memory when the mapped original memory was initialized. However, in the presence of type casting here and given that an `unsigned` value could be the start address of a memory region of any length, it is difficult to assure that the `memcpy` would just copy one byte. Thus, these detectors took a conservative approach of not updating the status of the associated shadow memory addresses in this situation. Consequently, when the variable `chip_id` was actually initialized through the casted pointer, the shadow memory of `chip_id` was still marked as *uninitialized*, causing the false positive.

To mitigate false positives caused so, these detectors may lift up or at least relax the conservative consideration (i.e., updating the status of the address `&chip_id` in the shadow memory after the call at Line 15). Generally this could possibly bring about false negatives (e.g., memory region after the address pointed to by `val`, at Line 5, which should not be marked as uninitialized but mistakenly done so hence the missed uninitialized vulnerabilities). Yet the risk may be practically acceptable—in this case, since exactly just one byte was specified to be copied (at Line 5),

it would not cause any false negatives if the status of `&chip_id` gets updated (to “initialized”) at Line 15.

The three purely dynamic detectors (VALGRIND, MEMORYSANITIZER and DRMEMORY) reported false vulnerabilities because they failed to deal with memory initializations through casted pointers, a challenge that can be mitigated through a less conservative memory status updating strategy.

```

1 typedef unsigned int __u32;
2 struct compstat {
3     __u32 unc_bytes ;
4     __u32 unc_packets ;
5     __u32 comp_bytes ;
6     __u32 comp_packets ;
7     __u32 inc_bytes ;
8     __u32 inc_packets ;
9     __u32 in_count ;
10    __u32 bytes_out ;
11    char ratio ;
12 };
13 int main() {
14     struct compstat cstats ;
15     cstats.ratio=(1<<0);
16     if(cstats.ratio)
17         printf("%d\n",cstats.ratio);
18     return 0;
19 }
```

Listing 8: In-depth inspection: Case 8.

Case 8. Listing 8 shows a positive sample, where a use of uninitialized variable error occurred at Line 16 since Line 15 only initialized the first bit of `cstats.ratio`. Only VALGRIND and MEMORYSANITIZER detected the error.

The reason was that the shadow memory mappings in the other two dynamic detectors DRMEMORY and ADDRESSSANITIZER were only byte-precise, compared to VALGRIND and MEMORYSANITIZER adopting bit-precise mapping. In CBMC, the assertion generation module treated the variable `cstats.ratio` as a whole rather than separately considering individual bits in the variable. Thus, no assertion for bit-precise checking was inserted during the static analysis, hence the false negative.

Intuitively, the coarse-precision memory checking may miss some memory errors that occur at higher-granularity locations. In practice, this byte-level precision may well suffice to find most of the relevant memory errors. Nevertheless, it is worth considering to raise the granularity level (to byte-precision checking), in light of our empirical findings (Section 4.3.2) indicating that a higher-precision detector (e.g., VALGRIND) may still be well scalable.

Only VALGRIND and MEMORYSANITIZER correctly reported the bit-precise *use of uninitialized variable* error, since they adopted bit-precise memory shadowing while others adopted a coarser-grained shadow memory mapping. A higher precision memory checking model can still be considered for better accuracy, without sacrificing efficiency much.

```

1 typedef unsigned char __u8;
2 typedef unsigned int __u32;
3 struct compstat {
4     __u8 unc_bytes ;
```

```

5     __u8 unc_packets ;
6     __u8 comp_bytes ;
7     __u8 comp_packets ;
8     __u8 inc_bytes ;
9     __u8 inc_packets ;
10    __u8 in_count ;
11    __u8 bytes_out ;
12 };
13 int main() {
14     struct compstat *cstats=malloc(sizeof(struct compstat));
15     __u32 *p=&(cstats->in_count);
16     *p=20;
17     free(cstats);
18 }
```

Listing 9: In-depth inspection: Case 9.

Case 9. Listing 9 shows a positive sample where an out-of-bounds error occurs at Line 16. The pointer `p` points to the start of `in_count`, a 8-bit field followed by another 8-bit field `bytes_out` of the `compstat` struct pointed to by `cstats`. Thus, starting at `p` is a 16-bit memory region. Yet Lines 15-16 attempt to write 20 to a 32-bit region that starts at `p`, hence an illegal access. All of the five detectors found this error but ADDRESSSANITIZER.

The reason was that ADDRESSSANITIZER only instrumented to check the starting byte of a variable of a built-in type, causing the false negative here with the partially out-of-bounds error. This issue was also mentioned in the original ADDRESSSANITIZER paper [35] where the authors clarified that they currently ignored such errors for better efficiency. In contrast, the other detectors checked every bit/byte for each memory access.

While this type of error is rare, byte-to-byte (or bit-to-bit) memory status mapping and checking would be helpful in avoiding the false negatives here. Meanwhile, as expected and pointed out in [35], this fine-grained analysis would compromise the efficiency of the detector.

ADDRESSSANITIZER did not capture partially out-of-bounds errors, since it only checked the starting byte of a built-in-type variable. A more fine-grained checking would mitigate this issue at the cost of lower efficiency.

4.5. RQ5: Impact of Benchmark Selection

We found that the source of program samples, as well as their code traits (e.g., loops and recursions) and data structures (e.g., arrays and embedded linked lists), had appreciable effects on the accuracy measures against the chosen detectors.

To understand the effects, let us first summarize how the two used datasets themselves differ. Notably, the entire *Software-Analyses-Benchmark* dataset was generated artificially. Most samples in this dataset are simple programs that just contain the vulnerability spot along with minimally necessary code contexts (e.g., variable declarations). In contrast, samples in the *SV-Benchmark* dataset were collected from diverse sources. Some samples in this dataset have complex data structures (e.g., samples of the *MemSafety_LinkedLists* category) or include complex code structures that are close to those seen in real-world

software systems (e.g., samples of the System_Busybox_MemSafety category).

Given these differences, ADDRESSSANITIZER, a hybrid analysis detector, had better F1 score (77.86%) than the two purely dynamic analysis detectors, VALGRIND (62.43%) and DRMEMORY (61.70%), against the *Software-Analysis-Benchmark* dataset. In comparison, ADDRESSSANITIZER had lower F1 score (57.14%) than VALGRIND (72.05%) and DRMEMORY (64.05%) against the *SV-Benchmark* dataset. For another example, CBMC, a purely static detector, had the best F1 score (87.31%) against the *Software-Analysis-Benchmark* dataset, while it had only 59.89% F1 score against the *SV-Benchmark* dataset, lower than VALGRIND and DRMEMORY.

The impact of the benchmark selection can be also seen between different categories in *SV-Benchmark* dataset. We found that the overall F1 score of these detectors against MemSafety_LinkedLists and System_Busybox_MemSafety was lower than other categories. Against System_Busybox_memsafety samples, three of these detectors had 0% F1 score, much lower than the accuracy they achieved against any other category. Note that the MemSafety_LinkedLists category is a sub-suite that includes samples that deal with complex data structures on stack or heap. The System_Busybox_MemSafety category includes samples that are parts of real software systems with memory safety issues, while most samples from other categories are relatively simple programs without complex logic and data structures. This reveals that detectors that had good performance against vulnerabilities in simple programs may still suffer relatively low performance against complex (e.g., real-world) software systems.

Beyond the impact of benchmark selection on the results of accuracy evaluation, the detectors also have seen that impact on efficiency results. With *Software-Analysis-Benchmark*, the dynamic detectors caused up to 1,000x runtime overhead. In comparison, with *SV-Benchmark*, the overhead was up to 50x. The reason was that the samples in *Software-Analysis-Benchmark* are very simple, thus the original program running time was commonly quite short—as a result, the total cost of checking a sample was easily many times of the original program running time. For another example, all of the five detectors finished detecting a single sample in *Software-Analysis-Benchmark* in less than 30 seconds on overall average, without any timeout cases. However, with the *SV-Benchmark* dataset, there were at most 32 timeout cases for a single detector and the maximum time cost of a single sample was greater than 200 minutes. In all, both the size and code complexity of the program samples considerably affected the efficiency of these detectors.

Finally, we note that the overall design intention of benchmark datasets naturally contributes to the impact of benchmark selection as well. One evidence in this regard in our study is the performance contrast between CBMC and the other four detectors. Intuitively, due to its underlying technique being a purely static analysis, CBMC was not

expected to be more effective. Yet it had the best overall accuracy against the *Software-Analysis-Benchmark* dataset. The reason, as mentioned earlier, was because this benchmark was designed for testing static vulnerability analysis techniques—more specifically, the samples were curated to include vulnerabilities that are relatively more detectable via static analysis.

Benchmark selection had clear impact on the efficiency and accuracy evaluation results of the studied detectors, and contributing to the impact are factors including code size, characteristics of code and data structures, code complexity, and design intention of the benchmark dataset overall.

5. Threats to Validity

Throughout our result analysis and discussion, we essentially referred to exchangeably a detector (e.g., AddressSanitizer) and the detection technique on which the detector is based (e.g., hybrid code analysis). A relevant threat is that the comparative evaluation results on the detection techniques covered by the chosen detectors may be different from what we reported, if different tool implementations of the same techniques were used. Our current study essentially treated each of the chosen detectors as a good representative of the respective underlying detection technique, which may have caused biases in our results.

In addition, our current categorization of detection techniques was relatively coarse-grained. We simply classified the five detectors into three high-level categories (static analysis, dynamic analysis, and hybrid analysis). Basically, our current categorization was not really based on the particular detection techniques (algorithms), but based on the *high-level classes* of the techniques (in terms of the types of program analysis on which the techniques are based). We did not differentiate specific detection algorithms within each category (e.g., different kinds of static-analysis-based detection techniques). For this reason, the evaluation in this study was indeed on the classes of detection techniques, rather than on different specific techniques in terms of detection algorithms.

Another threat lies in the representativeness of the chosen samples among actual C/C++ programs that users may use the chosen detectors against. It is commonly difficult to find samples that would represent all real-world programs. For this reason, we do not claim that our results and findings necessarily generalize to other programs. Thus, users who run the studied detectors against other real-world software applications may experience different performance results from what we presented.

To mitigate this threat, we made our best effort to account for the quality of the benchmark datasets used in our study, by using the datasets that have been used in prior peer studies and/or by other researchers and practitioners. For example, the chosen samples in the *SV-Benchmark*

dataset are used for a well-known annual competition event. These samples are not trivial. In fact, we found that many of these samples are complete real-world applications and many others are adapted from or part of real-world code. This at least provides confidence that the vulnerable code patterns are reasonably retained with respect to real-world software. We also purposely chose to use two quite different benchmark datasets, instead of one as in most prior peer studies, to further reduce possible biases. For a more comprehensive evaluation and comparison with our study, we need to use samples for which vulnerability ground truth is available, which is difficult to obtain for a sizable set of real-world applications. This is a key reason behind our choice of the samples that we chose.

Nevertheless, the reported results are best interpreted with respect to the samples we actually used in our study. The size and complexity of these samples may not have the same impact on the performance of the chosen detectors as would real-world applications. We also note that our results are mainly applicable to detectors focusing on memory error vulnerabilities that are based on static and/or dynamic code analysis. Relevant insights may not apply to code-analysis-based detectors targeting other classes of vulnerabilities, nor to detectors that are based on other techniques (e.g., data-driven approaches).

The distribution of positive/negative samples may also be a threat to the validity of the evaluation results [13]. In this study, the numbers of positive samples and those of negative samples used were almost balanced, which may not necessarily be consistent with the distribution in real-world applications. Ideally, the vulnerability distribution of each benchmark dataset should be as similar to the real-world vulnerability distribution as possible. However, it is unknown how vulnerable applications versus clean applications are actually distributed in the real world.

The vulnerabilities in the program samples were considered so based on the nature of the operation of the code, rather than their practical security consequences. Usually, people consider vulnerabilities based on the security consequences a software defect may cause. For example, the buffer overflow issue we discussed in Case 1 could have been treated as a functionality bug instead of a vulnerability. In this study, we treated all types of bugs (e.g., buffer overflow, use after free) that may potentially be used for malicious purposes as vulnerabilities, without considering the practical security consequences. One of the reasons is that our program samples are not all real-world software applications, and the reason we still chose to use these samples was again because of the availability of vulnerability ground truth with them.

Lastly, we used F1 score as an accuracy metric to evaluate and compare the detectors, which assumed precision and recall were equally important. This may not always be true in practice, where precision or recall might be valued more than the other. Thus, the accuracy in terms of the F1 score we reported may be different from what a user would actually experience with the studied detectors

(e.g., if the user emphasizes more on precision or recall).

6. Discussion

Based on our empirical findings, we derive insights and make recommendations on improving memory error vulnerability detection techniques based on code analysis.

Key lessons learned. Concerning detection accuracy, it turned out that none of the chosen detectors won over others against any kinds of vulnerabilities. As expected, each technique has its own strengths and weakness, related to not only the different focuses of the underlying technical design (e.g., performing a static check before run-time detection versus purely dynamic detection) but also to the varying complexity of the samples (e.g., containing deep loops or recursions versus using complicated data structures) the technique deals with.

In fact, it appears that precision and recall were competing goals for almost all of the techniques studied (see Figures 6 and 5)—it is generally difficult to achieve precision and recall that are both high. Instead, it was more of a tradeoff between the two. One example is that DRMEMORY suffered from false negatives (i.e., towards lower recall) due to the same strategy as that would help it reduce false positives (i.e., towards higher precision).

Intuitively, hybrid techniques would generally outperform purely static or dynamic techniques since the hybrid ones combine the merits of both static and dynamic analysis. This did not seem to be true for memory error vulnerability detection, as our results suggested. Meanwhile, dynamic techniques did not necessarily have higher precision than static ones, nor did static techniques necessarily have higher recall than dynamic ones. One main underlying factor is the specific detection strategy a technique actually chooses to adopt. For instance, this was demonstrated in Case 8 in our case studies.

Static techniques have both accuracy and efficiency advantages against simple programs, where all the control flow paths and assertions can be quickly traversed. Yet the resource usage would be substantial for large and complex programs. This was why CBMC consistently outperformed other detectors against the *Software-Analysis-Benchmark* dataset, while it did not and even had more timeout cases against the *SV-Benchmark* dataset.

The code traits of the samples themselves were clearly another factor that impacts the evaluation results. In fact, for RQ5, we have discussed the impact of benchmark selection on the evaluation results of the chosen detectors. If a benchmark dataset (e.g., *Software-Analysis-Benchmark*) was originally curated with a particular purpose in mind (e.g., for testing static analysis techniques), it would be intuitively biased in favor of tools that well fit the purpose (e.g., a static detector like CBMC that is purely based on static analysis). Considering such biases is necessary in interpreting evaluation results obtained from using such datasets.

Recommendations. Since no single particular techniques

consistently won and each demonstrated its specific merits and limitations, it is natural to consider integrating different techniques. However, a caveat here is that integration does not mean simple combinations of different types of code analysis—hybrid analyses may not be surely better than purely static or dynamic ones. For example, preceding dynamic detection with static, semantic code check that is aware of language semantics could bring higher accuracy than otherwise.

Meanwhile, developing a generally optimal code-analysis-based solution to vulnerability detection seems to be intractable. Thus, making *tradeoffs* may be inevitable, such as trading precision for recall or the other way around. Also, code features used in subject programs seem to have played a notable role in the accuracy of a technique. Thus, one may need to well understand the targeted kinds of programs in order to develop a more cost-effective technique. And given the challenge of making a technique work well for all programs, it would be rewarding to *prioritize* by targeting particular kinds of programs (e.g., those of particular code traits).

With respect to the common design of the studied dynamic detectors, a major concern regarding the vulnerability detection algorithm consists in the granularity of shadow memory. Intuitively, a finer granularity means more detailed monitoring of memory states hence more precise detection results. Meanwhile, finer-grained memory shadowing comes with the cost of higher memory consumption. Nevertheless, our empirical results suggested that even with the most fine-grained (i.e., bit-precise) mapping, the peak memory usage was still acceptable with respect to today’s memory capacity at a commodity machine. Thus, a viable recommendation for designing a precise dynamic detector would be to adopt a fine-grained shadow memory mapping mechanism for which the memory cost can be afforded on platforms where the detector is supposed to work.

7. Related Work

Prior works have addressed the comparisons of software vulnerability analysis techniques. For instance, several studies [3, 14, 4] have compared detection techniques that are based on different static code analysis approaches but all targeted SQL injection and XSS attack vulnerabilities in web services. In [23, 32, 39], the authors compared buffer-overflow vulnerability detectors. These earlier studies generally address a narrow scope of vulnerabilities and/or a specific application domains. Also, the evaluations and comparisons involved relatively small numbers of samples, and the vulnerability cases examined were limited to those detected by the studied tools—cases missed by the tools were not considered.

Austin et al. [7, 6] compared vulnerability detection techniques based on penetration testing and static analysis. Also, as in a few other studies [32, 3], the comparisons were done by counting the number of vulnerabilities found,

without referring to any ground truth. Thus, precision and recall were not rigorously computed in the comparisons. In [5], metrics for vulnerability detector benchmarking were provided without empirical experiments actually performed. Similarly, in [1, 39], capabilities of chosen vulnerability detectors were discussed comparatively, but only from technical perspectives and in an analytical fashion—no empirical comparisons were conducted. Thus, the actual detection performance of the compared tools were not assessed.

A few comparative studies [4, 14] performed empirical experiments, used vulnerability ground truth, and computed precision and recall, which differentiates them, just like our study separates itself, from all the prior peer work mentioned above. Yet in these two studies only commercial tools were addressed, as opposed to our study targeting open-source tools.

We recently conducted a preliminary study [30] of the same five open-source memory error vulnerability detectors against the *Software-Analysis-Benchmark* suite [37], addressing research questions corresponding to the first three in this paper. Since the benchmarks used were all manually crafted, which did not demonstrate statistically significant and large differences for most of the tool pairs compared (see Table 3). This paper subsumes that preliminary work and extends it in multiple ways. First, the scale of the study is doubled, with an entire new benchmark suite (i.e., the *SV-Benchmark*) considered additionally. This new benchmark suite is different from the originally used one in many aspects, including size, negative/positive sample distribution, complexity, and closeness to real-world applications, etc. Second, the scope of the study is largely expanded, by including two additional research questions beyond the original three. Also, for the original questions, results of larger amounts and discussions of greater depths are presented in this paper. Third, new content on background concepts and techniques along with more details on study design and justification have been added. Fourth, we now have dedicated, extensive discussion on validity threats and more insights as well as learned lessons and recommendations, which were not included in the prior work.

8. Conclusion

We conducted an extensive study on memory error vulnerability detection techniques through five state-of-the-art open-source tools in this domain against two different, carefully chosen sets of C/C++ program samples that cover 14 and 6 categories of memory error vulnerabilities, respectively. We assessed the performance of these memory error vulnerability detectors in terms of accuracy and efficiency metrics, and compared them through extensive statistical analyses. We also conducted a set of in-depth case studies to dissect the underlying causes of failures encountered by some of the detectors against certain samples. Our study revealed a number of new

findings on the status quo of code-analysis-based memory error vulnerability detection. These findings further enabled us to derive novel insights into the performance differences among the studied techniques, as well as to make actionable recommendations on future tool development for more effective vulnerability detection in terms of technical design choices and tradeoffs.

Acknowledgment

We thank the anonymous reviewers for their constructive comments. This research was sponsored by the Army Research Office and was accomplished under Grant Number W911NF-21-1-0027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Office or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

- [1] Amankwah, R., Kudjo, P.K., Antwi, S.Y., 2017. Evaluation of software vulnerability detection methods and tools: a review. International Journal of Computer Applications 169, 22–7.
- [2] Antunes, N., Laranjeiro, N., Vieira, M., Madeira, H., 2009. Effective detection of SQL/XPath injection vulnerabilities in web services, in: IEEE International Conference on Services Computing, IEEE. pp. 260–267.
- [3] Antunes, N., Vieira, M., 2009. 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, pp. 301–306.
- [4] Antunes, N., Vieira, M., 2010. Benchmarking vulnerability detection tools for web services, in: IEEE International Conference on Web Services, IEEE. pp. 203–210.
- [5] Antunes, N., Vieira, M., 2015. On the metrics for benchmarking vulnerability detection tools, in: 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE. pp. 505–516.
- [6] Austin, A., Holmgreen, C., Williams, L., 2013. A comparison of the efficiency and effectiveness of vulnerability discovery techniques. Information and Software Technology 55, 1279–1288.
- [7] Austin, A., Williams, L., 2011. One technique is not enough: A comparison of vulnerability discovery techniques, in: International Symposium on Empirical Software Engineering and Measurement, IEEE. pp. 97–106.
- [8] Beyer, D., 2012. Competition on software verification, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer. pp. 504–524.
- [9] Bruening, D., Zhao, Q., 2011. Practical memory checking with Dr. Memory, in: International Symposium on Code Generation and Optimization, IEEE. pp. 213–223.
- [10] Cai, H., Jenkins, J., 2018. Leveraging historical versions of android apps for efficient and precise taint analysis, in: Proceedings of the 15th International Conference on Mining Software Repositories, pp. 265–269.
- [11] Christakis, M., Bird, C., 2016. What developers want and need from program analysis: an empirical study, in: Proceedings of the 31st IEEE/ACM international conference on Automated Software Engineering, pp. 332–343.
- [12] Cliff, N., 1996. Ordinal methods for behavioral data analysis. Psychology Press.
- [13] Di Nucci, D., Palomba, F., Tamburri, D.A., Serebrenik, A., De Lucia, A., 2018. Detecting code smells using machine learning techniques: are we there yet?, in: IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, IEEE. pp. 612–621.
- [14] Fonseca, J., Vieira, M., Madeira, H., 2007. Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks, in: 13th Pacific Rim International Symposium on Dependable Computing, IEEE. pp. 365–372.
- [15] Fu, X., Cai, H., 2019. A dynamic taint analyzer for distributed systems, in: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1115–1119.
- [16] Ghaffarian, S.M., Shahriari, H.R., 2017. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. ACM Computing Surveys 50, 1–36.
- [17] Henning, J.L., 2006. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News 34, 1–17.
- [18] Jie, G., Xiao-Hui, K., Qiang, L., 2016. Survey on software vulnerability analysis method based on machine learning, in: IEEE First International Conference on Data Science in Cyberspace, IEEE. pp. 642–647.
- [19] Kals, S., Kirda, E., Kruegel, C., Jovanovic, N., 2006. Secubat: a web vulnerability scanner, in: Proceedings of the 15th International Conference on World Wide Web, pp. 247–256.
- [20] Kroening, D., Tautschig, M., 2014. CBMC-C bounded model checker, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer. pp. 389–391.
- [21] Lattner, C., Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE. pp. 75–86.
- [22] Li, G., Gopalakrishnan, G., 2010. Scalable SMT-based verification of gpu kernel functions, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM. pp. 187–196.
- [23] Li, P., Cui, B., 2010. A comparative study on software vulnerability static analysis techniques and tools, in: International Conference on Information Theory and Information Security, pp. 521–524.
- [24] Li, W., Cai, H., Sui, Y., Manz, D., 2020. PCA: memory leak detection using partial call-path analysis, in: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1621–1625.
- [25] Milicevic, A., Kugler, H., 2011. Model checking using smt and theory of lists, in: Nasa Formal Methods Symposium, Springer. pp. 282–297.
- [26] Nethercote, N., Seward, J., 2007a. How to shadow every byte of memory used by a program, in: Proceedings of the 3rd International Conference on Virtual Execution Environments, ACM. pp. 65–74.
- [27] Nethercote, N., Seward, J., 2007b. Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan notices 42, 89–100.
- [28] Newsome, J., Song, D., 2005. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software, in: Proceedings of the 12th Network and Distributed Systems Security Symposium, Citeseer.
- [29] Nicolau, A., 1985. Loop quantization: unwinding for fine-grain parallelism exploitation. Technical Report. Cornell University.
- [30] Nong, Y., Cai, H., 2020. A preliminary study on open-source memory vulnerability detectors, in: International Conference on Software Analysis, Evolution, and Reengineering (SANER), pp. 557–561.
- [31] Petukhov, A., Kozlov, D., 2008. Detecting security vulnerabilities in web applications using dynamic analysis with penetration testing. Computing Systems Lab, Department of Computer Science, Moscow State University , 1–120.
- [32] Pozza, D., Sisto, R., Durante, L., Valenzano, A., 2006. Comparing lexical analysis tools for buffer overflow detection in network software, in: 1st International Conference on Communication Systems Software & Middleware, IEEE. pp. 1–7.

- [33] Rasool, A., 2019. Which is the most vulnerable programming language? <https://www.digitalinformationworld.com/2019/03/searching-for-the-most-secure-programming-language.html>. Accessed on October 24, 2019.
- [34] Romano, J., Kromrey, J.D., Coraggio, J., Skowronek, J., Devine, L., 2006. 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. pp. 1–51.
- [35] Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D., 2012. AddressSanitizer: A fast address sanity checker, in: USENIX Annual Technical Conference, pp. 309–318.
- [36] Seward, J., Nethercote, N., 2005. Using Valgrind to detect undefined value errors with bit-precision., in: USENIX Annual Technical Conference, General Track, pp. 17–30.
- [37] Shiraiishi, S., Mohan, V., Marimuthu, H., 2015. Test suites for benchmarks of static analysis tools, in: International Symposium on Software Reliability Engineering Workshops, pp. 12–15.
- [38] Shirey, R., 2000. Internet security glossary.
- [39] Silberman, P., Johnson, R., 2004. A comparison of buffer overflow prevention implementations and weaknesses. IDEFENSE, August .
- [40] Stepanov, E., Serebryany, K., 2015. MemorySanitizer: fast detector of uninitialized memory use in C++, in: IEEE/ACM International Symposium on Code Generation and Optimization, IEEE. pp. 46–55.
- [41] Vieira, M., Antunes, N., Madeira, H., 2009. Using web security scanners to detect vulnerabilities in web services, in: IEEE/IFIP International Conference on Dependable Systems & Networks, IEEE. pp. 566–571.
- [42] Walpole, R.E., Myers, R.H., Myers, S.L., Ye, K.E., 2011. Probability and Statistics for Engineers and Scientists. Prentice Hall.