Detection of Security Vulnerabilities in C Code using Runtime Verification: an Experience Report

Kostyantyn Vorobyov, Nikolai Kosmatov, and Julien Signoles

CEA, LIST, Software Reliability and Security Laboratory,
PC 174, 91191 Gif-sur-Yvette France
{kostyantyn.vorobyov,nikolai.kosmatov,julien.signoles}@cea.fr

Abstract. Despite significant progress made by runtime verification tools in recent years, memory errors remain one of the primary threats to software security. The present work is aimed at providing an objective up-to-date experience study on the capacity of modern online runtime verification tools to automatically detect security flaws in C programs. The reported experiments are performed using three advanced runtime verification tools (E-ACSL, Google Sanitizer and RV-Match) over 700 test cases belonging to SARD-100 test suite of the SAMATE project and Toyota ITC Benchmark, a publicly available benchmarking suite developed at the Toyota InfoTechnology Center. SARD-100 specifically targets security flaws identified by the Common Weakness Enumeration (CWE) taxonomy, while Toyota ITC Benchmark addresses more general memory defects, as well as numerical and concurrency issues. We compare tools based on different approaches – a formal semantic based tool, a formal specification verifier and a memory debugger – and evaluate their cumulative detection capacity.
The results of the experiments indicate that the selected tools cumilatively detected 84% of all seeded defects. Although for several categories of errors detection rates are higher, we observed that applying several tools is beneficial for uncovering certain issues. For instance, in detecting concurrency issues of the Toyota ITC Benchmark, the highest per-tool result was 73%, whereas cumulative detection ratio of all three tools used together was 93%.

Keywords: Runtime Verification · Software Security · Memory Safety · Dynamic Analysis · Experience Report

1 Introduction

The C programming language is one of the most commonly used languages for development of critical software such as operating systems, drivers, hypervisors, cryptography libraries, etc. At the same time C lacks protection mechanisms, leaving the entire responsibility for correct management of memory and resources to the developer. Execution of a badly written C program can lead to an *undefined behaviour*, one that is not formally specified by the C language standard

[21]. Undefined behaviours in C programs are potential causes of *memory errors*, such as buffer overflows, format string vulnerabilities, double free violations and many others. Memory errors are known to be one of the main threats to software security [40] because they can be exploited by an attacker to trigger execution of malicious code capable of stealing or corrupting data, provoking system failures and even taking control of the affected machine. Detecting memory errors is therefore of the utmost importance for security of code.

One way to automatically detect memory errors in a program is by using static program analysis techniques, such as abstract interpretation [9], deductive methods [13] and model checking [16,26]. An orthogonal approach to detecting memory errors is by means of dynamic analysis, for instance, using online runtime verification [17], a technique that observes an execution of the program and detects errors before they occur. Over recent years online runtime verification (sometimes referred to as *monitoring*) has been successfully used to detect numerous undefined behaviours in C programs. For instance, over 300 previously unknown errors have been detected in the Chromium browser using AddressSanitizer [29].

Static analysis techniques analyze programs without executing them, but typically require fine-tuning to get usable results (including a low number of false alarms for abstract interpretation, proved properties for deductive methods and termination in a reasonable amount of time for model checking). In contrast to static methods, runtime verification analyzes programs by executing them in a concrete setting, eliminating the need for fine-grained tuning and reducing engineering effort to get proper results. One of the drawbacks of runtime verification, however, is that it analyzes one behaviour at a time. This is different to static analysis techniques that typically analyze all program behaviours. It is also possible to run the program in a simulated environment. The benefit of this approach is that it does not need a concrete setting or extensive tuning. Simulated runs, however, typically suffer from significant performance overheads.

Motivation. Due to increasing interest in verification techniques and tools in academia and industry they rapidly evolve, offering novel solutions or significant improvements almost every year. As such, related experimental studies tend to become outdated very fast. Most of such studies are performed by the authors of a tool and aimed at comparing it with its previous version or a few related tools in order to demonstrate the benefits of the proposed solution. Such studies do not provide a global view of the state of the art: for instance, such an important research question as the *cumulative detection capacity* of several tools used together is often not addressed.

More importantly, the main focus of experiments with monitoring tools is about to change. Since the execution overhead used to be an important barrier to their application, many prior experimental studies [42,41,29,37,31,4] focused primarily on tools' performance and often conducted experiments using computationally-intensive programs rather than investigated the detection power. However, due the increasing computation power of modern computers

and the recent progress of monitoring techniques, the execution overhead does not necessarily represent critical limitations for many monitoring tools today.

Finally, software security has become one of the most important concerns in many domains of software engineering and automatic detection of potential security vulnerabilities is an attractive and promising application for monitoring tools. Both researchers and practitioners need to have an objective and up-to-date vision of what kinds of security vulnerabilities can be detected by modern state-of-the-art tools tools and which ones are likely to remain undetected.

The Runtime Verification Competitions [2,18] (held in the context of the International Conference on Runtime Verification since 2014) evaluate both soundness and performance of the participating tools. These events are very helpful for the tool developers allowing them to compare their tools with others and identify their weaknesses. However, the competitions cannot provide a complete global picture either. This is because the participating tools are typically evaluated on a handful of benchmarks submitted by the authors of the tools. Further, these competitions do not focus on security vulnerabilities.

Goals and Means. The purpose of this work is to provide an experience report evaluating the capacity of bug checking tools to detect security vulnerabilities with an emphasis on memory errors. We choose the C programming language as one of the most relevant languages for security-critical software. Our first objective is to give an *up-to-date vision* of this detection capacity at a large scale.

To reach this goal, we selected two existing publicly available benchmark suites with over 700 test cases in total. They are representative of security-related vulnerabilities and already classified into several subcategories. The first benchmark we used is SARD-100 [11], a test suite belonging to the Software Assurance Metrics And Tool Evaluation (SAMATE) project of the National Institute of Standards and Technology (NIST). SARD-100 targets security flaws of the Common Weakness Enumeration (CWE) taxonomy [8]. The second suite, Toyota ITC Benchmark [34], is a publicly available benchmarking suite developed at the Toyota InfoTechnology Center. Toyota ITC Benchmark mostly focusses on memory errors, but also contains a number of programs seeded with numerical and concurrency issues.

Evaluating tools of such a large number of test cases requires tool automation. We therefore choose to run the selected tools using a fixed set of documented options (when required) without any specific tuning. This approach allows us to give a *realistic and objective picture* of how effective the compared tools can be when used by a competent engineer who is not a developer of the tool. This approach particularly fits security vulnerability detection: in this context, most end-users favour automation to precision. Since static tools require such a fine-tuning, we exclude them for our experimentation and focus on runtime verification tools only. We selected a representative subset of online monitoring tools (E-ACSL, Google Sanitizer and RV-Match) following a number of well-defined

criteria such as *availability*, *robustness*, capacity of *online monitoring*[1]. Additionally, we try to consider different scientific approaches – a formal semantic based tool, a formal specification verifier and a memory debugger – in order to objectively evaluate the cumulative detection power of these tools used together.

As we have mentioned above, previous studies often focused on tool performance that is not a critical limitation for many monitoring tools today. Also, including tool performance in our evaluation would be unfair for RV-Match which is based on a simulated environment. Therefore, we also exclude performance evaluation from our study in order to focus only on detection capability. This way, it allows us to provide a *global and objective view* of how effective modern runtime verification tools can be for verification engineers looking for security vulnerabilities.

Contributions. The contributions of this paper include:

- an experimental campaign aimed at evaluating the capacity to detect security-related vulnerabilities using three modern runtime verification tools, E-ACSL, Google Sanitizer and RV-Match, and two benchmarking suites, SARD-100 and Toyota ITC Benchmark;
- an experience report presenting the recorded results separately for each (sub-)category and each tool, as well as globally over the three tools and for all programs; detailed results of each tool for each benchmark program are available in the companion report[2];
- a careful analysis of the reported results showing where we stand with runtime detection of security vulnerabilities using monitoring tools.

Outline. The rest of the paper is organized as follows. Section 2 presents related work on runtime monitoring techniques and tools. Section 3 describes our experimental setup and explains the choice of selected tools and benchmarks. Section 4 presents and discusses the experimental results, and Section 5 summarizes the threats to validity of the present experiment. Finally, Section 6 presents concluding remarks and future work directions.

2 Related Work

We now review techniques that focus on runtime detection of memory errors in C programs.

Online monitoring of C programs for memory-related software vulnerabilities goes decades back. One of the oldest (yet still active) techniques to detecting defects in C programs at runtime is Rational Purify [20]. This tool instruments object files with additional instructions that track memory state of an executing program and identify operations on unallocated or uninitialized memory locations. More recent techniques, such as MemCheck [31], SGCheck [32], or Dr.

[1] which dynamically analyzes a program run on the fly, unlike *offline monitoring* based on previously recorded execution traces.

[2] See `https://goo.gl/S4NF5m`

Memory [4] achieve a similar task using dynamic binary instrumentation, an approach that injects memory monitors to binary programs at execution time.

One of the benefits of binary-level memory monitoring is its ability to check every memory access (typically using memory shadowing) including those occurring in C library or third-party code. Binary monitors are widely used during development and testing stages in many large-scale software projects [39]. However, since such tools reason at the level of instructions they often fail to detect issues visible only at a source level of the language including such problems as misuse of pointers, type violations or use of variable addresses outside of scope of their definition[3].

Source-level techniques to runtime detection of memory errors have also been developed. Seminal work of Jones and Kelly [23] enabled runtime bounds checking using a splay tree to track program pointers and bounds of objects they reference. At runtime this technique checks operations on pointers (e.g., dereference, arithmetic) by querying the splay tree-shaped metadata. This technique served as a building block for a number of runtime memory error detectors. CRED [28] added support for tracking out-of-bounds pointers using an additional auxiliary hash table. Dhurjati and Adve [12] improved bounds checking technique using Automatic Pool Allocation memory partitioning. Baggy bounds checking [1] used specialized memory allocator to constrain size and alignment of allocated blocks and used array-based lookup to improve performance. MemSafe [35] used a mix of static and dynamic analyses to prevent memory errors at runtime via a combination of object and pointer metadata.

Even though effective for tracking memory errors the above techniques have not gone beyond research prototypes, as such they are difficult to use in practice without expert knowledge.

Google AddressSanitizer [29] is probably the first successful attempt to create a widely-used industrial-strength source-level monitor for C/C++ programs. AddressSanitizer uses shadow memory to track program allocations at runtime using source-to-source transformations. This tool benefits from a compact shadow state encoding that tracks 8-byte sequences by only 3 bits. This allows for significant reduction of monitoring overhead costs. Initially integrated with the clang compiler AddressSanitizer has later been ported to GCC replacing mudflap [14] tool. Nowadays AddressSanitizer is a part of a bigger tool suite called Google Sanitizer that contains several tools that focus on different issues: AddressSanitizer (illegal memory accesses and memory leaks), MemorySanitizer [37] (uninitialized memory accesses), ThreadSanitizer [30] (data races and deadlocks) and UndefinedBehaviourSanitizer [38] (undefined behaviours).

One of the disadvantages of fully automatic analyzers such as AddressSanitizer or MemCheck is that they cannot easily enforce custom properties (e.g., function contracts or loop invariants). Such issues can be addressed using behaviour interface specification languages such as E-ACSL. E-ACSL [10] is a run-

[3] More precisely, in some cases memory corruption errors caused by such violations are detectable by binary analysis tools but only after they are disconnected from the source code error that caused them.

time verification tool for C programs built atop the Frama-C [24] framework for source-code analysis. E-ACSL transforms a C program P annotated with formal specifications in the E-ACSL specification language into a monitored program P' that behaves similar to P but aborts at runtime if any given annotation is violated. Formal E-ACSL specifications usable by the tool can be provided manually or generated automatically by another tool such as the Frama-C kernel or its RTE plug-in [24] The present focus of E-ACSL is runtime enforcement of function contracts, detection of integer overflows and validating memory accesses made by the program at runtime.

An orthogonal way of detecting errors (including security vulnerabilities) in C programs is by using simulated environments. One such tool is RV-Match [19]. The aim of RV-Match is to ensure that a run of a C program strictly conforms to the ISO C11 [5] standard, i.e., does not rely on implementation specific or undefined behaviours described by the standard.

RV-Match is built using the $\mathbb{K}$ semantics framework [27]. $\mathbb{K}$ is a program analysis framework based on term rewriting that allows to define rigorous semantics for a target programming language. The framework also provides several tools for formal analysis of programs written in the target language including a symbolic execution engine, a semantic debugger, a model checker and a deductive verifier. RV-Match uses formal executable C semantics [15] to instantiate the $\mathbb{K}$ framework for C and interprets programs according to the formal operational semantics of the language.

Another approach to enforcing memory safety of C programs is called Cyclone [22]. Cyclone is a safe dialect of the C programming language designed to retain C semantics and performance and at the same time prevent memory-related errors. To achieve this goal Cyclone imposes restrictions on C programs. For instance, Cyclone limits pointer arithmetic, enforces pointer initialization and disallows unsafe casts. This approach also uses "fat-pointers" to enable runtime bounds checks and prevent accesses to unallocated memory. Presently Cyclone is no longer supported but some of its ideas made into the Rust programming language [25] that pursues similar goals.

Overall, technique such Cyclone or Rust are compromises between safety and security and performance that prevent many issues commonly associated with C programs by design. Using such techniques for an existing program, however, may be a daunting task as it requires porting a program to one of these languages.

3 Experimental Setup

3.1 Objectives and Evaluation Approach

The key objective of this paper is to evaluate the capacity of state-of-the-art monitoring tools to detect security vulnerabilities in C programs. We address this objective using an empirical study that analyses benchmarked code belonging to the test suite #100 of the Software Assurance Reference Dataset Project (SARD) [11] and the Toyota InfoTechnology Center dataset [6,34] using E-ACSL [10], Google Sanitizer [29,37,30,38] and RV-Match [19]. For each tool

we compute its detection ratio (i.e. the number of discovered bugs over the total number of bugs) and report the results.

More precisely, this evaluation seeks answers to the following research questions:

(RQ1) What is the cumulative detection ratio of the selected state-of-the-art tools used together for each category of vulnerabilities?
(RQ2) What is the detection ratio of each of the selected tools for each category of vulnerabilities?
(RQ3) Are different tools complementary in the bugs they detect?

The following sections provide details on the choice of the tools and benchmarks used in this empirical study and discusses evaluation methodology.

3.2 Selected Tools

We now discuss the key selection criteria of the runtime verification tools used in the present experiment.

Availability and Robustness One of our goals is to evaluate runtime verification techniques usable by most developers. For this experimentation we select freely available, robust tools capable of verifying C code at runtime with no or little manual effort. Consequently we reject research prototypes, incomplete implementations, or techniques usable only by experts.

Memory Analysis Many vulnerabilities in C occur due to its almost unrestricted use of memory. Therefore we only consider tools capable of analysing the memory state of a running program. Another source of security flaws in C programs are executions that lead to undefined behaviours with respect to a chosen ISO C standard. Since such issues are defined at a source level of the C programming language we only consider tools using source code analysis. Consequently we reject binary monitors such as MemCheck [31] or Dr Memory [4].

Online Monitoring In a security-oriented analysis it is important to prevent errors, as otherwise a vulnerability can be exploited before it can be reported. To address this requirement for this experimentation we consider only online runtime verification tools capable of detecting vulnerabilities before they occur.

Potential Complementarity To have a global vision of the cumulative detection capacity of different state-of-the-art techniques, we select tools using different approaches to runtime error detection. In other words, we reject tools that approach a problem similarly but differ in implementation.

Based on the above requirements for this experimentation we select the following tools (in alphabetic order):

E-ACSL [10] – a verifier of a rich specification language;
Google Sanitizer [29,37,30,38] – a source-level memory debugger;
RV-Match [19] – a verifier of formal language semantics.

More detailed descriptions of the selected tools is given in Section 2.

3.3 Selected Benchmarks

We now briefly discuss the source code benchmarks used in this empirical study.

SARD-100 [11] is a test suite belonging to the Software Assurance Metrics And Tool Evaluation (SAMATE) project of the National Institute of Standards and Technology (NIST). Each SARD-100 program contains a vulnerability of the Common Weakness Enumeration (CWE) taxonomy [8]. Initially developed for testing against source code security analyzers based on Source Code Security Analysis Tool Functional Specification [3] SARD-100 explores such important security issues as SQL and command injections, buffer overflows, format string vulnerabilities, use-after-free errors and others (21 CWE vulnerabilities in total).

Toyota ITC Benchmark [6,34] is a publicly available benchmarking suite developed at Toyota InfoTechnology Center, USA. The suite is based on Annex A (Source Code Weaknesses) of Source Code Security Analysis Tool Functional Specification [3]. Toyota ITC Benchmark consists of 638 test cases exploring 9 defect types and 51 sub-types. Toyota ITC Benchmark focusses on memory defects (e.g., static, dynamic, stack, pointer arithmetic), numerical defects (such as division by zero or integer overflows) and concurrency issues (race conditions, deadlocks).

One of the key factors for selecting SARD-100 and Toyota ITC Benchmark for the present experimentation is that both contain code samples originating from reliable sources with clearly marked vulnerabilities. The test cases belonging to these suites allow to explore a broad range of security-related issues typical to C programs.

3.4 Evaluation Methodology

During the present experimentation we perform a series of program runs under E-ACSL, Google Sanitizer and RV-Match monitoring and compute detection ratio of each tool. The percent detection ratio of a tool run over programs containing N defects is computed as $D/N * 100$, where D is the number of defects detected by the analyzer.

In this experiment we used latest stable versions of the tools available at the time of the experiment [4]. The RV-Match and Google Sanitizer monitored programs were obtained via `kcc-1.0` and `clang-4.0.1` compilers respectively. Since these tools make use of built-in analyses no external specifications were provided. Programs monitored via E-ACSL (version 0.9) were obtained via its driver script called `e-acsl-gcc.sh` that takes a C program, automatically annotates it using the RTE [24] plugin of Frama-C and finally compiles it using the `gcc` compiler (`gcc` version 5.4.0 was used). For E-ACSL analysis we also used partial function contracts provided by the Frama-C standard library.

During this experimentation a defect is considered detected if it is reported either during compile stage or before its occurrence at runtime. We consider runs

[4] At the time of this writing (March, 2018) E-ACSL-0.9 is not available publicly yet and was obtained from the developers of the tool. E-ACSL-0.9 is scheduled to be released in May, 2018.

of monitors that failed due to internal errors or by intercepting signals as missing defects. Also, since Google Sanitizer consists of 4 separate tools (AddressSanitizer, UndefinedBehaviourSanitizer, ThreadSanitizer and MemorySanitizer) we consider a defect detected if it is reported by either of the tools.

The monitored programs were run once per tool except for the cases using random number generators that potentially surpass the execution of vulnerable code. In such cases the benchmarks were run continuously until erroneous paths were explored.

The tools used in this experiment were run using inputs provided via the benchmarking suites. Where such inputs were not available (several programs from SARD-100) we used inputs that explored vulnerabilities in the benchmarked code.

The platform for all results reported here was 2.30GHz Intel i7 processor with 16GB RAM, running 64-bit Gentoo Linux.

False positives Even though some of the tools used in this experimentation can produce false alarms (in particular AddressSanitizer [29]), we do not report false positive detection rate for the tools. This is because no false alarms were detected during this experimentation. In other words, for this particular study the rate of false positive defects equates to 0% in all cases. We manually verified that all defects reported by the tools during this study correspond to actual defects.

Runtime Overheads In dynamic analysis performance overhead of a technique is an important issue, however, this experimentation does not report runtime or memory overheads of the tools. This is because of the following reasons.

The goal of this experimentation is in identifying a technique's capability to detect different security-related issues rather than evaluating its applicability to large and computationally intensive programs. Consequently, the experiment uses small programs whose execution time in most cases does not exceed one second. The size of the programs used during this experiment is not representative for evaluating the tools' robustness with respect to performance overhead.

Performance overhead of the tools used in this experimentation was assessed in prior work. For instance, [42] reports on runtime and memory overheads of both E-ACSL and AddressSanitizer using computationally intensive benchmarks for CPU testing developed by the Standard Performance Evaluation Corporation (SPEC CPU) [36]. This experimentation has shown that for the selected SPEC CPU programs the runtime overheads of E-ACSL (on average 19 times compared to execution of unmonitored programs) of AddressSanitizer (1.58 times). Further, the experiment has shown that E-ACSL was was more memory efficient comparing to AddressSanitizer (2.48 vs. 4.22 times on average).

Thorough empirical study evaluating performance overheads of RV-Match has not been conducted. This is mainly because at the present stage of implementation RV-Match is not yet ready to deal with large programs. A preliminary result reported in [19] shows that analysis of a small example program consisting of 10,000 loop iterations took 11 seconds. Taking into account that an unobserved execution of the same program is under 0.01 seconds, such a result

	E-ACSL	Sanitizer	RV-Match	Cumulative
Non-memory Defects				
CWE-078: Command Injection	0% (0/6)	0% (0/6)	0% (0/6)	0% (0/6)
CWE-080: Basic XSS	0% (0/5)	0% (0/5)	0% (0/5)	0% (0/5)
CWE-089: SQL Injection	0% (0/4)	0% (0/4)	0% (0/4)	0% (0/4)
CWE-099: Resource Injection	0% (0/4)	0% (0/4)	0% (0/4)	0% (0/4)
CWE-259: Hard-coded Password	0% (0/5)	0% (0/5)	0% (0/5)	0% (0/5)
CWE-489: Leftover Debug Code	0% (0/1)	0% (0/1)	0% (0/1)	0% (0/1)
Memory Defects				
CWE-121: Stack Buffer Overflow	100% (11/11)	91% (10/11)	91% (10/11)	100% (11/11)
CWE-122: Heap Buffer Overflow	100% (6/6)	100% (6/6)	100% (6/6)	100% (6/6)
CWE-416: Use After Free	100% (9/9)	100% (9/9)	100% (9/9)	100% (9/9)
CWE-244: Heap Inspection	0% (0/1)	0% (0/1)	0% (0/1)	0% (0/1)
CWE-401: Memory Leak	100% (5/5)	80% (4/5)	60% (3/5)	100% (5/5)
CWE-468: Pointer Scaling	50% (1/2)	50% (1/2)	50% (1/2)	50% (1/2)
CWE-476: Null Dereference	100% (7/7)	100% (7/7)	100% (7/7)	100% (7/7)
CWE-457: Uninitialized Variable	100% (4/4)	75% (3/4)	100% (4/4)	100% (4/4)
CWE-415: Double Free	100% (6/6)	100% (6/6)	67% (4/6)	100% (6/6)
CWE-134: Format String	100% (8/8)	0% (0/8)	0% (0/8)	100% (8/8)
CWE-170: String Termination	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
CWE-251: String Management	100% (5/5)	100% (5/5)	100% (5/5)	100% (5/5)
CWE-391: Unchecked Error	0% (0/1)	0% (0/1)	0% (0/1)	0% (0/1)
Concurrency Defects				
CWE-367: Race Condition	0% (0/4)	0% (0/4)	0% (0/4)	0% (0/4)
CWE-412: Unrestricted Lock	0% (0/1)	0% (0/1)	0% (0/1)	0% (0/1)
Overall	67% (67/100)	56% (56/100)	54% (54/100)	67% (67/100)

Table 1. Detection results of E-ACSL, Google Sanitizer and RV-Match over SARD-100 test suite

might suggest that overheads of RV-Match are presently too high to be used in practice with real-world programs.

4 Experimental results

We now discuss detection results of using E-ACSL, Google Sanitizer and RV-Match over SARD-100 dataset and Toyota ITC Benchmark.

4.1 Results for SARD-100

Table 1 shows detection results of E-ACSL, Google Sanitizer and RV-Match over C benchmarks belonging to the SARD-100 dataset. The leftmost column of the table shows a given CWE vulnerability, the rest of the columns show error detection ratio in percent followed by the number of discovered and overall defects.

The overall results indicate that E-ACSL, Google Sanitizer and RV-Match detected 67%, 56% and 54% of bugs respectively. Cumulative error detection ratio (i.e., with respect to defects detected by at least one of the tools) is 67%.

All tools missed security vulnerabilities that do not lead to memory errors (*Non-memory defects* category), namely resource, command and SQL injections, cross-site scripting and issues related to the use of hard-coded passwords. Such result is because the focus of both Google Sanitizer and RV-Match is analysis aiming to detect memory errors and undefined behaviors. The executions exploring these vulnerabilities did not lead to such issues.

The results further indicate that all tools are well equipped for detection of memory-related errors including buffer overflows, null pointer dereferences, use-after-free and similar issues. During analysis of memory-related defects (*Memory Defects* category) E-ACSL detected all seeded defects except for one test case utilizing bad input to `scanf` (*CWE-391*). Google Sanitizer and RV-Match detected less issues. For instance, in *CWE-121* both tools missed a defect that uses a fixed size buffer to store user-supplied input via `puts` function. Furthermore, RV-Match does not support detection of heap memory leaks when memory is allocated via library functions (such as `strdup`). We should also note that Google Sanitizer missed one memory leak because this tool does not treat memory that has not been freed but available via a global variable as a leak. Finally, Google Sanitizer and RV-Match do not support runtime detection of format string vulnerabilities (via standard library functions such as `printf`).

Even though Google Sanitizer and RV-Match include functionality allowing to discover concurrency issues (e.g., race conditions), both tools missed a handful of such defects in SARD-100. E-ACSL does not support monitoring of multi-threaded programs.

4.2 Results for Toyota ITC Benchmark

Table 2 shows detection results of E-ACSL, Google Sanitizer and RV-Match over programs from Toyota ITC Benchmark. The presentation of the results is similar to that of Table 1 in Section 4.1.

The results indicate that E-ACSL, Google Sanitizer and RV-Match have detected 71%, 57% and 84% of defects respectively. Cumulative error detection ratio over Toyota ITC Benchmark is 87%.

All tools detected most defects related to improper use of dynamic and static memory that include such issues as buffer over- and underflows (top two rows of Table 2). 78% detection rate in *Dynamic* defects of Google Sanitizer is because this tool could not detect a number of heap buffer-underruns where negative offsets were used to access unallocated memory.

In *Pointer-related* defects E-ACSL detected 56% of bugs. The tool could not identify defects related to improper use of function pointers. Detection ratio of Google Sanitizer is 32%. This result is the lowest of all tools in the *Pointer-related* category. This tool had issues detecting defects related to passing a null pointer to `free`, incorrect pointer arithmetic and the use of function pointers.

Defect Type	E-ACSL	Sanitizer	RV-Match	Cumulative
Dynamic Memory	94% (81/86)	78% (67/86)	94% (81/86)	94% (81/86)
Static Memory	100% (67/67)	96% (64/67)	100% (67/67)	100% (67/67)
Pointer-related	56% (47/84)	32% (27/84)	99% (83/84)	99% (83/84)
Stack-related	35% (7/20)	70% (14/20)	100% (20/20)	100% (20/20)
Resource	99% (95/96)	60% (58/96)	98% (94/96)	100% (96/96)
Numeric	93% (100/108)	59% (64/108)	98% (106/108)	98% (106/108)
Miscellaneous	94% (33/35)	49% (17/35)	71% (25/35)	97% (34/35)
Inappropriate Code	0% (0/64)	0% (0/64)	0% (0/64)	0% (0/64)
Concurrency	0% (0/44)	73% (32/44)	66% (29/44)	93% (41/44)
Overall	71% (430/604)	57% (343/604)	84% (505/604)	87% (530/604)

Table 2. Detection results of E-ACSL, Google Sanitizer and RV-Match over Toyota ITC Benchmark

RV-Match detected 99% of errors related to improper use of pointers. This tool missed only one defect related to using an uninitialized pointer.

Stack-related defect type includes three defect sub-types: stack overflow, cross-thread access and static buffer overrun. E-ACSL has little support for detecting stack overflows and cannot monitor multi-threaded programs. The detection rate of E-ACSL in this case is only 35%. Google Sanitizer and RV-Match provide better support and identified 70% and 100% of defects respectively.

Resource Management defects of Toyota ITC Benchmark contain such issues as double free, freeing non-dynamic memory, return of local addresses and memory leaks. For this vulnerability type Google Sanitizer has the lowest detection ratio of 60%. This tool has missed several bugs related to freeing static memory and returning local variables. E-ACSL on the contrary has been able to identify most such defects (99%). RV-Match has similar result of 98%.

E-ACSL and RV-Match detected most errors of *Numeric* defect type (integer overflows, bit-field overflows, division by zero), 93% and 98%. Both tools missed defects involving floating point overflow. Additionally, E-ACSL failed to detect overflows via bit-field values. Google Sanitizer has detected fewer defects (59% via UndefinedBehaviourSanitizer). This tool had issues detecting errors involving integer precision loss because of cast and loss of integer sign because of unsigned casts.

Miscellaneous defects of Toyota ITC Benchmark describe endless loops, invalid extern variable declarations, missing return statements and similar issues. E-ACSL detected 94% of such issues. Detection ratio for Google Sanitizer and RV-Match were lower – 49% and 71% respectively.

All tools missed all defects of the *Inappropriate Code* type. Such issues consist of mostly syntactic and stylistic issues such as left-over debug code, specifying

same condition twice and so on. Even though they contribute to the overall score such defects do not lead to undefined behaviours or memory errors and can hardly be regarded as security-related.

Finally, E-ACSL missed all defects of *Concurrency* type. As noted earlier E-ACSL does not support detection of such issues. Google Sanitizer and RV-Match detected 73% and 66% respectively. A notable result is that these tools miss defects of different subtypes and the detection cumulative rate over both tools is 93%.

4.3 Summary of Results

The overall results of our experiment show that the present state-of-the-art runtime verification tools for C programs provide strong support for detection of issues stemming from improper use of memory and undefined behaviours. On the other hand the results indicate lack of support for such important security flaws as command injections that remain one of the most dangerous software bugs [7] and still found even in well-tested C applications [33].

With respect to **RQ1** that considers the cumulative capacity of error detection, we can conclude that in tracking non-concurrent memory errors using one tool may be sufficient to detect most defects. This is supported by the results for SARD-100 and Toyota ITC Benchmark that show consistently high detection ratios for typical memory errors such as buffer overflows, double free violations, null pointers dereferences, use of initialized values and so on. Regarding **RQ3**, the results indicate that combining the results of all tools one can achieve higher detection ratio. For instance, in detection of resource-management memory defects of Toyota ITC Benchmark, E-ACSL detected 99% of all defects yet the cumulative detection ratio is 100%. Further, while E-ACSL detected 94% defects of miscellaneous type of Toyota ITC Benchmark, the combined result (with RV-Match) is 97%. The most interesting result is for detection of concurrency issues. Notably, the detection ratios of both Google Sanitizer and RV-Match are relatively low – 73% and 66%, cumulative result however is 93%, almost all seeded defects of that type.

Per-tool analysis of results (see **RQ2**) shows that for detection of memory-related vulnerabilities in single-threaded executions E-ACSL shows superior performance. One reason for such result is the tool's memory tracking model that allows tracking bounds of memory blocks and identify more vulnerabilities involving illegal memory accesses. E-ACSL is also the only tool that enables runtime analysis for format string vulnerabilities. On the other hand both Google Sanitizer and RV-Match show good scores for detecting defects in multi-threaded executions (e.g., deadlocks and race conditions). E-ACSL has no support for such analysis. Furthermore, both tools have also shown better support for detecting improper use of function pointers and stack overflows.

5 Threats to Validity

We now discuss issues that might have affected validity of the experiment presented in this paper.

The first issue refers to the choice of source code benchmarks used in evaluating precision of runtime verification tools. We aimed to select representative code covering a broad range of defects typically leading to security vulnerabilities in C. Different choice of programs might affect the results. For instance, neither SARD-100 nor Toyota ITC Benchmark explores issues related to information flow leakage in its full generality. Since the analyzers used during the experiment have no support for such analysis the precision results of all tools could be lower if such issues were present.

Another issue refers to the choice of the runtime verification tools used in the experiment. We aimed to select popular online monitoring tools capable of detecting typical security issues occurring in C program with little manual effort. However, we cannot claim that our tool selection was representative for detecting security vulnerabilities. For instance, as shown by the experiment E-ACSL, Google Sanitizer and RV-Match have good support for detecting defects related to the use of memory and undefined behaviours but do not support detection of SQL or command injections. Furthermore, we might have overlooked some of the tools that address similar issues and also freely available and easy to use.

The third issue refers to generating monitored programs using Google Sanitizer. This tool has a number of compile- and runtime options affecting the results of its analysis. Even though we tried to study this tool and use all documented features there might be some options that we overlooked. Furthermore, Google Sanitizer is under active development and we used the latest released version. It is therefore possible that the latest development version of Google Sanitizer has a different precision over the same code samples.

The fourth issue refers to collecting results. During our experimentation we discovered several bugs in programs from SARD-100 and Toyota ITC Benchmark. To ensure the correctness of the precision results we manually verified that programs under analysis are correct (i.e., contain defects of the claimed types and the provided inputs lead to execution of erroneous path). However, as we had to deal with large number of defects (over 700) we might have overlooked some of the issues that might have also affected the final results.

Finally, the authors of this paper have been involved in design and the development of the E-ACSL runtime verification tool. While we did our best to stay impartial and aimed at providing a fair and unbiased study we might have had the *developers' advantage* when reasoning about the results produced by E-ACSL.

6 Conclusions and Future Work

This experience report provides a global view of the capacity of modern runtime verification tools to detect security vulnerabilities with an emphasis on memory errors. We consider different approaches – a formal semantic based tool, a formal specification verifier and a memory debugger – in order to evaluate the cumulative detection power of these tools used together. We have presented the

experimental protocol, the selected tools and benchmarks, and provided and analyzed the recorded results. Detailed results are available online and can be used for a more detailed analysis. They indicate the level of support by the selected tools for various kinds of issues. Overall, the cumulative detection rate of the three selected tools over all defects of the considered benchmark suites is 84%. Although detection rates achieve highest values for several categories of errors, we observed that applying several tools appears to be beneficial for detecting several categories of issues. For instance, in detecting concurrency issues in the Toyota ITC Benchmark the highest per-tool result is 73%, whereas the cumulative rate is 93%. Future work includes experiments with other categories of tools, using larger benchmark suites of security related issues, as well as further analysis of failures and improvement of their support in the compared tools.

Acknowledgments The authors thank the Frama-C team for providing the tools and support. Many thanks to the anonymous referees for their helpful comments.

References

1. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the USENIX Security Symposium. pp. 51–66. USENIX Association (August 2009)
2. Bartocci, E., Bonakdarpour, B., Falcone, Y.: First international competition on software for runtime verification. In: Proceedings of the International Conference on Runtime Verification. LNCS, vol. 8734, pp. 1–9. Springer (2014)
3. Black, P.E., Kass, M., Koo, M., Fong, E.: Source code security analysis tool functional specification version 1.1. Tech. Rep. 500-268 v1.1, Information Technology Laboratory (2011)
4. Bruening, D., Zhao, Q.: Practical memory checking with Dr. Memory. In: Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 213–223. CGO '11, IEEE Computer Society, Washington, DC, USA (2011)
5. ISO/IEC 9899:2011, `https://www.iso.org/standard/57853.html`
6. Center, T.I.: Specification of test bench (2015), `https://github.com/regehr/itc-benchmarks`
7. Christey, S.: 2011 CWE/SANS top 25 most dangerous software errors. Tech. Rep. 1.0.3, The MITRE Corporation, `http://www.mitre.org` (September 2011)
8. Common Weakness Enumeration: A community developed dictionary of software weakness types, `http://cwe.mitre.org`
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Symposium on Principles of Programming Languages (POPL'77). pp. 238–252 (1977)
10. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Proceedings of the ACM Symposium on Applied Computing. pp. 1230–1235. ACM (March 2013)
11. Delaitre, A.: Test Suite #100: C test suite for source code analyzer v2 - vulnerable (2015), `https://samate.nist.gov/SRD/view.php?tsID=100`
12. Dhurjati, D., Adve, V.S.: Backwards-compatible array bounds checking for C with very low overhead. In: Proceedings of the International Conference on Software Engineering. pp. 162–171. ACM (May 2006)

13. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
14. Eigler, F.C.: Mudflap: pointer use checking for C/C++. In: Proceedings of the GCC Developers Summit. pp. 57–70 (May 2003)
15. Ellison, C., Rosu, G.: An executable formal semantics of C with applications. SIGPLAN Notices 47(1), 533–544 (January 2012)
16. Emerson, E.A., Clarke, E.M.: Characterizing correctness properties of parallel programs using fixpoints. Automata, Languages and Programming 85/1980 (1980)
17. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems, pp. 141–175 (2013)
18. Falcone, Y., Nickovic, D., Reger, G., Thoma, D.: Second international competition on runtime verification CRV 2015. In: Proceedings of the International Conference on Runtime Verification. LNCS, vol. 9333, pp. 405–422. Springer (2015)
19. Guth, D., Hathhorn, C., Saxena, M., Roşu, G.: RV-Match: Practical semantics-based program analysis. In: Proceedings of the International Conference on Computer Aided Verification. LNCS, vol. 9779, pp. 447–453. Springer (July 2016)
20. Hastings, R., Joyce, B.: Purify: Fast detection of memory leaks and access errors. In: Proceedings of the Winter USENIX Conference. pp. 125–136 (January 1992)
21. ISO/IEC 9899:1999, 1430 Broadway, New York, NY 10018, USA: Programming languages – C (March 2013), `www.open-std.org/jtc1/sc22/wg14/www/standards`
22. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: Proceedings of the General Track: 2002 USENIX Annual Technical Conference. pp. 275–288. USENIX (June 2002)
23. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proceedings of the International Workshop on Automatic Debugging. pp. 13–26. Linköping University Electronic Press (September 1997)
24. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (2015)
25. Matsakis, N.D., Klock, II, F.S.: The Rust language. Ada Letters 34(3), 103–104 (October 2014)
26. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: International Symposium on Programming. LNCS, vol. 137, pp. 337–351. Springer (1982)
27. Rosu, G., Serbanuta, T.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
28. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the Network and Distributed System Security Symposium. The Internet Society (December 2004)
29. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A fast address sanity checker. In: Proceedings of the USENIX Annual Technical Conference. pp. 309–319. USENIX Association (June 2012)
30. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic race detection with LLVM compiler — compile-time instrumentation for ThreadSanitizer. In: Proceedings of the International Conference on Runtime Verification. LNCS, vol. 7186, pp. 110–114. Springer (September 2011)
31. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the USENIX Annual Technical Conference. pp. 17–30. USENIX (2005)

32. SGCheck: an experimental stack and global array overrun detector. http://valgrind.org/docs/manual/sg-manual.html
33. Vulnerability summary for CVE-2014-6271. National Institute of Standards and Technilogy: National Vulnerability Database (September 2014), http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271
34. Shiraishi, S., Mohan, V., Marimuthu, H.: Test suites for benchmarks of static analysis tools. In: IEEE International Symposium on Software Reliability Engineering Workshops. pp. 12–15. IEEE (2015)
35. Simpson, M.S., Barua, R.: MemSafe: ensuring the spatial and temporal memory safety of C at runtime. Software: Practice and Experience 43(1), 93–128 (2013)
36. Standard Performance Evaluation Corporation: SPEC CPU (2006), http://www.spec.org/benchmarks.html
37. Stepanov, E., Serebryany, K.: MemorySanitizer: fast detector of uninitialized memory use in C++. In: Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 46–55. IEEE Computer Society (February 2015)
38. Undefinedbehaviorsanitizer (2017), https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html
39. Projects using Valgrind. http://valgrind.org/gallery/users.html (2017)
40. van der Veen, V., dutt-Sharma, N., Cavallaro, L., Bos, H.: Memory errors: The past, the present, and the future. In: Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses. LNCS, vol. 7462, pp. 86–106. Springer (September 2012)
41. Vorobyov, K., Kosmatov, N., Signoles, J., Jakobsson, A.: Runtime detection of temporal memory errors. In: Proceedings of the International Conference on Runtime Verification. pp. 294–311. Springer (September 2017)
42. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: Proceedings of the International Symposium on Memory Management. pp. 47–58. ACM (June 2017)