CAV'16, LNCS 9779, pp 447-453. 2016

RV-Match: Practical Semantics-Based Program Analysis

Dwight Guth[1], Chris Hathhorn[1,3], Manasvi Saxena[1,2], and Grigore Roşu[1,2]

[1] Runtime Verification Inc., Urbana, USA
{dwight.guth,chris.hathhorn}@runtimeverification.com
{manasvi.saxena,grigore.rosu}@runtimeverification.com
[2] University of Illinois at Urbana-Champaign, Urbana, USA
[3] University of Missouri, Columbia, USA

Abstract. We present RV-Match, a tool for checking C programs for undefined behavior and other common programmer mistakes. Our tool is extracted from the most complete formal semantics of the C11 language. Previous versions of this tool were used primarily for testing the correctness of the semantics, but we have improved it into a tool for doing practical analysis of real C programs. It beats many similar tools in its ability to catch a broad range of undesirable behaviors. We demonstrate this with comparisons based on a third-party benchmark.

Keywords: C11, programming language semantics, undefined behavior, static analysis, abstract interpretation.

1 Introduction

The $\mathbb{K}$ semantic framework[4] is a program analysis environment based on term rewriting [1]. Users define the formal semantics of a target programming language and the $\mathbb{K}$ framework provides a series of formal analysis tools specialized for that language, such as a symbolic execution engine, a semantic debugger, a systematic checker for undesired behaviors (model checker), and even a fully fledged deductive program verifier. Our tool, RV-Match, is based on the $\mathbb{K}$ framework instantiated with the publicly-available C11 semantics[5] [6, 7], a rigorous formalization of the current ISO C11 standard [10]. We have specially optimized RV-Match for the execution and detection of errors in C programs.

Unlike modern optimizing compilers, which have a goal to produce binaries that are as small and as fast as possible at the expense of compiling programs that may be semantically incorrect, RV-Match instead aims at mathematically rigorous dynamic checking of programs for strict conformance with the ISO C11 standard. A strictly-conforming program is one that does not rely on implementation-specific behaviors and is free of the most notorious feature of the C language, *undefined behavior*. Undefined behaviors are semantic holes left by the standard for implementations to fill in. They are the source of many subtle bugs and security issues [9].

[4] http://kframework.org

[5] https://github.com/kframework/c-semantics

CAV'16, LNCS 9779, pp 447-453. 2016

Running RV-Match. Users interface with RV-Match through the `kcc` executable, which behaves as a drop-in replacement for compilers like `gcc` and `clang`. Consider a file `undef.c` with contents:

```
int main(void) {
      int a;
      &a+2; }
```

We compile the program with `kcc` just as we would with `gcc` or `clang`. This produces an executable named `a.out` by default, which should behave just as an executable produced by another compiler—for strictly-conforming, valid programs. For undefined or invalid programs, however, `kcc` reports errors and exits if it cannot recover:

```
$ kcc undef.c
$ ./a.out
Error: UB-CEA1
Description: A pointer (or array subscript) outside the
  bounds of an object.
Type: Undefined behavior.
See also: C11 sec. 6.5.6:8, J.2:1 item 46
  at main(undef.c:2)
```

In addition to location information and a stack trace, `kcc` also cites relevant sections of the standard [10].

2 Practical Semantics-Based Program Analysis

Unlike similar tools, we do not instrument an executable produced by a separate compiler. Instead, RV-Match directly interprets programs according to a formal operational semantics. The semantics gives a separate treatment to the three main phases of a C implementation: compilation, linking, and execution. The first two phases together form the "translation" semantics, which we extract into an OCaml program to be executed by the `kcc` tool. The `kcc` tool, then, translates C programs according to the semantics, producing an abstract syntax tree as the result of the compilation and linking phases. This AST then becomes the input to another OCaml program extracted from the execution semantics.

The tool on which we have based our work was originally born as a method for testing the correctness of the operational semantics from which it was extracted [7], but the performance and scalability limitations of this original version did not make it a practical option for analysis of real programs. To this end, we have improved the tool on several fronts:

– *OCaml-based execution engine.* We implemented a new execution engine that interprets programs according to a language semantics 3 orders of magnitude faster than our previous Java-based version. For this improvement in performance, we take advantage of the optimized pattern-matching implemented by the OCaml compiler, a natural fit for $\mathbb{K}$ Framework semantics.

In the course of this work, we uncovered and fixed a few limitations of the OCaml compiler itself in dealing with very large pattern match expressions. [6]

- *Native libraries.* Previous versions of our tool required all libraries to be given semantics (or their C source code) before they could be interpreted. We now support linking against and calling native libraries, automatically marshalling data to and from the representation used in the semantics.
- *Expanded translation phase.* In our C semantics, we now calculate the type of all terms, the values of initializers, and generally do more evaluation of programs during the translation phase. Previously, much of this work was duplicated during execution.
- *Error recovery and implementation-defined behavior.* We have implemented error recovery and expanded support for implementation-defined behavior. Programs generated by older versions of `kcc` would halt when encountering undefined or implementation-defined behavior. Our new version of `kcc` gives semantics for many common undefined behaviors so the interpreter can continue with what was likely the expected behavior after reporting the error. Similarly, we have added support for implementation profiles, giving users an easy way to parameterize the semantics over the behaviors of common C implementations.
- *Scope of errors.* We have also expanded the breadth of the errors reported by `kcc` to include bad practices and errors involving standard library functions.[7]

These improvements have allowed `kcc` to build and analyze programs in excess of 300k lines of code, including the BIND DNS server.

Performance evaluation. For an idea of the extent of the performance enhancements over previous versions of our tool, consider this simple program that calculates the sum of integers between 0 and 10000:

```
#include <stdio.h>
int main(void) {
      int i, sum = 0;
      for (i = 0; i < 10000; ++i) sum += i;
      printf("Sum: %d\n", sum); }
```

In the table below, we compare the time in seconds to compile and run this program five times with an old version of our tool[8] [9] to our new version using our OCaml execution engine. The first and second rows report the average time for five compilations and runs,[9] respectively, and the third reports the sum of all

[6] See `http://caml.inria.fr/mantis/view.php?id=6883` and `http://caml.inria.fr/mantis/view.php?id=6913`.

[7] For a summary of the kinds of errors `kcc` will report, see `https://github.com/kframework/c-semantics/blob/master/examples/error-codes/Error_Codes.csv`.

[8] Version 3.4.0, with $\mathbb{K}$ Framework version 3.4.

[9] These tests were run on a dual CPU 2.4GHz Intel Xeon with 8GB of memory. On more memory-intensive programs, we see an additional order of magnitude or more improvement in performance.

runs plus the average compilation time to simulate the case of a compiled test being run on different input.

	Old `kcc`	New `kcc`	Change
Avg. compile time	13 s	2 s	−85%
Avg. run time	816 s	11 s	−99%
All runs + avg. comp.	4092 s	59 s	−99%

3 Evaluation

Of course, many other tools exist for analyzing C programs. In this section, we compare RV-Match with some popular C analyzers on a benchmark from Toyota ITC. We also briefly mention our experience with running our tool on the SV-COMP benchmark. The other tools we consider:

- *GrammaTech CodeSonar* is a static analysis tool for identifying "bugs that can result in system crashes, unexpected behavior, and security breaches" [8].
- *MathWorks Polyspace Bug Finder* is a static analyzer for identifying "run-time errors, concurrency issues, security vulnerabilities, and other defects in C and C++ embedded software" [11].
- *MathWorks Polyspace Code Prover* is a tool based on abstract interpretation that "proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code" [12].
- *Clang UBSan, TSan, MSan, and ASan (version 3.7.1)* are all `clang` modules for instrumenting compiled binaries with various mechanisms for detecting undefined behavior, data races, uninitialized reads, and various memory issues, respectively [5].
- *Valgrind Memcheck and Helgrind (version 3.10.1, GCC version 4.8.4)* are tools for instrumenting binaries for the detection of several memory and thread-related issues (illegal reads/writes, use of uninitialized or unaddressable values, deadlocks, data races, etc.) [13].
- *The CompCert C interpreter (version 2.6)* uses an approach similar to our own. It executes programs according to the semantics used by the CompCert compiler [3] and reports undefined behavior.
- *Frama-C Value Analysis (version sodium-20150201)*, like Code Prover, is a tool based on static analysis and abstract interpretation for catching several forms of undefinedness [4].

The Toyota ITC benchmark [14]. This publicly-available[10] benchmark consists of 1,276 tests, half with planted defects meant to evaluate the defect rate capability of analysis tools and the other half without defects meant to evaluate the false positive rate. The tests are grouped in nine categories: static memory, dynamic memory, stack-related, numerical, resource management, pointer-related, concurrency, inappropriate code, and miscellaneous.

[10] `https://github.com/Toyota-ITC-SSD/Software-Analysis-Benchmark`

We evaluated RV-Match along with the tools mentioned above on this benchmark. Our results appear in Figure 1 and the tools we used for our evaluation are available online.[11] Following the method of Shiraishi, Mohan, and Marimuthu [14], we report the value of three metrics: DR is the detection rate, the percentage of tests containing errors where the error was detected; $\underline{\mathsf{FPR}} = 100 - \mathsf{FPR}$, where FPR is the false positive rate; and PM is a productivity metric, where $\mathsf{PM} = \sqrt{\mathsf{DR} \times \underline{\mathsf{FPR}}}$, the geometric mean of DR and $\underline{\mathsf{FPR}}$.

Interestingly, and similar to our experience with the SV-COMP benchmark mentioned below, the use of RV-Match on the Toyota ITC benchmark detected a number of flaws in the benchmark itself, both in the form of undefined behavior that was not intended, and in the form of tests that were intended to contain a defect but were actually correct. Our fixes for these issues were accepted by the Toyota ITC authors and we used the fixed version of the benchmark in our experiments. Unfortunately, we do not have access to the MathWorks and GrammaTech static analysis tools, so in Figure 1 we have reproduced the results reported in Shiraishi, Mohan, and Marimuthu [14]. Thus, it is possible that the metrics scored for the other tools may be off by some amount.

The SV-COMP benchmark suite. This consist of a large number of C programs used as verification tasks during the International Competition on Software Verification (SV-COMP) [2]. We analyzed 1346 programs classified as correct with RV-Match and observed that 188 (14%) of the programs exhibited undefined behavior. Issues ranged from using uninitialized values in expressions, potentially invalid conversions, incompatible declarations, to more subtle strict aliasing violations. Our detailed results are available online.[12]

4 Conclusion

We have presented RV-Match, a semantics-based ISO C11 compliance checker. It does better than the other tools we considered in terms of its detection rate, and note that it reports *no false positives*. Also, we think our experience with finding undefined behavior even in the presumed-correct programs of the above benchmarks demonstrates our tool's usefulness.

We do not claim, however, that our approach is simply better than the approaches represented by the other tools. We see our technology as a complement to other approaches. Static analysis tools, for example, are more forgiving in terms of analyzing code that does not even compile, so they can help find errors earlier. They also typically analyze all code in one run of the tool. On the other hand, our tool, like all tools performing dynamic analysis, generally requires the program to actually execute in order to detect most errors. Our tool also limits itself to the code that is actually executed, so it is best combined with existing testing infrastructure (e.g., by running unit tests with `kcc`).

[11] `https://github.com/runtimeverification/evaluation/tree/master/toyota-itc-benchmark`

[12] `https://github.com/runtimeverification/evaluation/tree/master/svcomp-benchmark`

Tool		Static memory	Dynamic memory	Stack-related	Numerical	Resource management	Pointer-related	Concurrency	Inappropriate code	Misc.	Avg. (unweighted)	Avg. (weighted)
RV-Match (kcc)	DR	100	94	100	96	93	98	67	*0*	63	79	82
	FPR	100	100	100	100	100	100	100	–	100	100	100
	PM	100	97	100	98	96	99	82	*0*	79	89	91
GrammaTech CodeSonar	DR	100	89	*0*	48	61	52	70	46	69	59	68
	FPR	100	100	–	100	100	96	77	99	100	97	98
	PM	100	94	*0*	69	78	71	73	67	83	76	82
MathWorks Bug Finder	DR	97	90	15	41	55	69	*0*	28	69	52	62
	FPR	100	100	85	100	100	100	–	94	100	98	99
	PM	98	95	36	64	74	83	*0*	51	83	71	78
MathWorks Code Prover	DR	97	92	60	55	20	69	*0*	1	83	53	53
	FPR	100	95	70	99	90	93	–	97	100	94	95
	PM	98	93	65	74	42	80	*0*	10	91	71	71
UBSan + TSan + MSan + ASan (clang)	DR	79	16	95	59	47	58	67	*0*	37	51	47
	FPR	100	95	75	100	96	97	72	–	100	93	95
	PM	89	39	84	77	67	75	70	*0*	61	69	67
Valgrind + Helgrind (gcc)	DR	9	80	70	22	57	60	72	2	29	44	42
	FPR	100	95	80	100	100	100	79	100	100	95	97
	PM	30	87	75	47	76	77	76	13	53	65	65
CompCert interpreter	DR	97	29	35	48	32	87	58	17	63	52	51
	FPR	82	80	70	79	83	73	42	83	71	74	76
	PM	89	48	49	62	52	80	49	38	67	62	63
Frama-C Value Analysis	DR	82	79	45	79	63	81	7	33	83	61	66
	FPR	96	27	65	47	46	40	100	63	49	59	55
	PM	89	46	54	61	54	57	26	45	63	60	60

Fig. 1. Comparison of tools on the 1,276 tests of the ITC benchmark. The numbers for the GrammaTech and MathWorks tools come from Shiraishi, Mohan, and Marimuthu [14].

- Highlighting indicates the best score in a category for a particular metric.
- DR, FPR, and PM are, respectively, the detection rate, 100 − FPR (the complement of the false positive rate), and the productivity metric.
- The final average is weighted by the number of tests in each category.
- Italics and a dash indicate categories for which a tool has no support.

CAV'16, LNCS 9779, pp 447-453. 2016

References

[1] Grigore Roşu and Traian Florin Şerbănuţă. "An Overview of the K Semantic Framework". In: *J. Logic and Algebraic Programming* 79.6 (2010), pp. 397–434. DOI: 10.1016/j.jlap.2010.03.012.

[2] Dirk Beyer. "Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference (TACAS'16)". In: ed. by Marsha Chechik and Jean-François Raskin. 2016. Chap. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016), pp. 887–904. ISBN: 9783662496749. DOI: 10.1007/978-3-662-49674-9_55.

[3] Brian Campbell. "An Executable Semantics for CompCert C". In: *Certified Programs and Proofs*. Vol. 7679. Lecture Notes in Computer Science. Springer, 2012, pp. 60–75. DOI: 10.1007/978-3-642-35308-6_8.

[4] Géraud Canet, Pascal Cuoq, and Benjamin Monate. "A Value Analysis for C Programs". In: *Conf. on Source Code Analysis and Manipulation (SCAM'09)*. IEEE, 2009, pp. 123–124. DOI: 10.1109/SCAM.2009.22.

[5] Clang. *Clang 3.9 Documentation*. URL: http://clang.llvm.org/docs/index.html.

[6] Chucky Ellison. "A Formal Semantics of C with Applications". PhD thesis. University of Illinois, July 2012. URL: http://hdl.handle.net/2142/34297.

[7] Chucky Ellison and Grigore Roşu. "An Executable Formal Semantics of C with Applications". In: *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*. 2012, pp. 533–544. DOI: 10.1145/2103656.2103719.

[8] GrammaTech. *CodeSonar*. URL: http://grammatech.com/products/codesonar.

[9] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the Undefinedness of C". In: *36th Conf. on Programming Language Design and Implementation (PLDI'15)*. 2015.

[10] ISO/IEC JTC 1, SC 22, WG 14. *ISO/IEC 9899:2011: Prog. Lang.—C*. Tech. rep. Intl. Org. for Standardization, 2012.

[11] MathWorks. *Polyspace Bug Finder*. URL: http://www.mathworks.com/products/polyspace-bug-finder.

[12] MathWorks. *Polyspace Code Prover*. URL: http://www.mathworks.com/products/polyspace-code-prover.

[13] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, 2007, pp. 89–100. DOI: 10.1145/1250734.1250746.

[14] Shinichi Shiraishi, Veena Mohan, and Hemalatha Marimuthu. "Test Suites for Benchmarks of Static Analysis Tools". In: *The 26th IEEE International Symposium on Software Reliability Engineering (ISSRE'15)*. Vol. Industrial Track. 2015.