Combining Static Analysis and Test Generation for C Program Debugging

Omar Chebaro[1,2], Nikolai Kosmatov[1], Alain Giorgetti[2,3], and Jacques Julliand[2]

[1] CEA, LIST, Software Safety Laboratory, PC 94, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] LIFC, University of Franche-Comté, 25030 Besançon Cedex France
firstname.lastname@lifc.univ-fcomte.fr
[3] INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy France

Abstract. This paper presents our ongoing work on a tool prototype called SANTE (Static ANalysis and TEsting), implementing a combination of static analysis and structural program testing for detection of run-time errors in C programs. First, a static analysis tool (Frama-C) is called to generate alarms when it cannot ensure the absence of run-time errors. Second, these alarms guide a structural test generation tool (PathCrawler) trying to confirm alarms by activating bugs on some test cases. Our experiments on real-life software show that this combination can outperform the use of each technique independently.
Keywords: all-paths test generation, static analysis, run-time errors, C program debugging, alarm-guided test generation.

1 Introduction

Software validation remains a crucial part in software development process. Software testing accounts for about 50% of the total cost of software development. Automated software validation is aimed at reducing this cost. The increasing demand has motivated much research on automated software validation. Two major techniques have improved in recent years, dynamic and static analysis. They arose from different communities and evolved along parallel but separate tracks. Traditionally, they were viewed as separate domains.

Static analysis examines program code and reasons over all possible behaviors that might arise at run time. Since program verification is in general undecidable, it is often necessary to use approximations. Static analysis is conservative and sound: the results may be weaker than desirable, but they are guaranteed to generalize to all executions. Dynamic analysis operates by executing a program and observing this execution. It is in general incomplete due to a big (or even infinite) number of possible test cases. Dynamic analysis is efficient and precise because no approximation or abstraction needs to be done: the analysis can examine the actual, exact run-time behavior of the program for the corresponding test case.

The pros and cons of the two techniques are apparent. If dynamic analysis detects an error then the error is real. However, it cannot in general prove the absence of errors. On the other hand, if static analysis reports a potential error, it may be a false alarm. However, if it does not find any error (of a particular kind) in the overapproximation of program behaviors then the analyzed program clearly cannot contain

such errors. Static and dynamic analysis have complementary strengths and weaknesses and can be both applied to program verification. Static analysis is typically used for proofs of correctness. Dynamic analysis demonstrates the presence of errors and increases confidence in a system.

Recently, there has been much interest in combining dynamic and static methods for program verification [1–6]. Static and dynamic analyses can enhance each other by providing valuable information that would otherwise be unavailable. This paper reports on an ongoing project that aims to provide a new combination of static analysis and structural testing of C programs. We implement our method using two existing tools: Frama-C, a framework for static analysis of C programs, and PathCrawler, a structural test generation tool.

Frama-C [7] is being developed in collaboration between CEA LIST and the ProVal project of INRIA Saclay. Its software architecture is plug-in-oriented and allows fine-grained collaboration of analysis techniques. Static analyzers are implemented as plug-ins and can collaborate with one another to examine a C program. Frama-C is distributed as open source with various plug-ins. Developed at CEA LIST, PathCrawler [8] is a test generation tool for C functions respecting *the all-paths criterion,* which requires to cover all feasible program paths, or *the k-path criterion,* which restricts the generation to the paths with at most k consecutive iterations of each loop.

Contributions. This paper presents our ongoing work combining static analysis and structural test generation for validation of C programs, in particular, for detection of run-time errors. We call this technique *alarm-guided test generation.* Our ongoing implementation of this method, called SANTE, assembles two heterogeneous tools using quite different technologies (such as abstract interpretation and constraint logic programming). We evaluate our method by several experiments on real-life C programs, and compare the results with static analysis alone, test generation alone, and test generation guided by the exhaustive list of alarms for all potentially threatening statements. In all cases, our method outperforms the use of each technique independently.

The paper is organized as follows. Section 2 gives an overview of our method and its implementation in progress. Section 3 presents initial experiments illustrating the benefits of our approach. Section 4 briefly presents related work and concludes.

2 Overview of the Method

This section presents our method combining static analysis and test generation, and its ongoing implementation in a tool prototype SANTE[4] (Static ANalysis and TEsting) which uses Frama-C and PathCrawler tools. Our implementation choice was to connect PathCrawler and Frama-C via a new plug-in, and adapt PathCrawler to accept information provided by other plug-ins.

Algorithm 1 shows an overview of the method. SANTE takes as input the C program P to be analyzed and the test context (denoted by *Context*) defining the function to be analyzed, domains of its input variables and preconditions. We denote by $\alpha_i, i \in I$ the statements of the program P. SANTE starts by analyzing the program with

[4] The French word “santé” means “health”, and sometimes also “cheers!”

the value analysis plug-in of Frama-C. Based on abstract interpretation, this plug-in computes and stores supersets of possible value ranges of variables at each statement of the program. Among other applications, these over-approximated sets can be used to exclude the possibility of a run-time error. The value analysis is sound: it emits an alarm for an operation whenever it cannot guarantee the absence of run-time errors for this operation. It starts from an entry point in the analyzed program specified by the user, and unrolls function calls and loops. It memorizes abstract states at each statement and provides an interface for other plug-ins to extract these states.

The abstract states make it possible to extract $\Psi = \{\Psi_i \mid i \in I\}$, where Ψ_i is the condition restricting the state before the statement α_i to an error state, in other words, describing the states leading to a possible run-time error at α_i. For instance, for the statement `x=y/z;` the plug-in emits "*Alarm:* `z` *may be 0!*" and returns $\Psi_i \equiv (\mathtt{z} = 0)$ if 0 is contained in the superset of values computed for `z` before this statement. For the last statement in `int t[10]; ... t[n]=15;` the plug-in emits "*Alarm:* `t+n` *may be invalid!*" and returns $\Psi_i \equiv (\mathtt{n} < 0 \vee \mathtt{n} > 9)$ when it cannot exclude the risk of out-of-range index `n`. For `int* p; ... *(p+j)=10;` the plug-in emits "*Alarm:* `p+j` *may be invalid!*" if it cannot guarantee that `p+j` refers to a valid memory location. In the current version, the extraction of Ψ_i is supported for division by 0 and out-of-range array index, and not yet fully supported for invalid pointers or non-initialized variables. If value analysis sees no risk of a run-time error at α_i, then $\Psi_i \equiv$ *false*.

If all $\Psi_i \equiv$ *false*, i.e. no alarms were reported, then all possible program executions are error-free and the program is proved to contain no run-time errors. If some Ψ_i is not trivial, we use the following technique called *alarm-guided test generation* (lines 5–6 in SANTE). We realize a specific instrumentation of P represented here by the function ADDERRORBRANCHES. It takes as inputs the original program P and the alarms $\Psi_i, i \in I$ for its statements, and returns a new program $P' = \{\alpha'_i \mid i \in I\}$. ADDERRORBRANCHES iterates over the statements α_i of P and, if there is no alarm for α_i, keeps $\alpha'_i = \alpha_i$. Otherwise it replaces the statement α_i by the statement

`if(` Ψ_i `) storeBugAndExit(); else` α_i

In other words, if the alarm condition is verified, a run-time error can occur, so the function `storeBugAndExit()` reports a potential bug and stops the execution of the current test case. If there is no risk of run-time error, the execution continues normally and P' behaves exactly as P.

Next, PathCrawler is called for P'. The PathCrawler test generation method [8] is similar to the so-called *concolic,* or *dynamic symbolic execution*. The user provides the C source code of the function under test. The generator explores program paths in a depth-first search using symbolic and concrete execution. The transformation of P into P' adds new branches for error and error-free states so that the PathCrawler test generation algorithm will automatically try to cover error states. It returns the list of detected bugs B with error paths and inputs which confirms some alarms. Other alarms may remain unconfirmed due to various reasons: (1) this is a false alarm, (2) test generation timed/spaced out or (3) incomplete test selection strategy was used e.g. k-path.

Algorithm 1 Algorithm of the method

SANTE(P, *Context*)

1: Ψ := VALUEANALYSIS(P, *Context*)
2: **if** $\forall i \in I, \Psi_i \equiv$ *false* **then**
3: **return** *proved* /* no alarms */
4: **else**
5: P' := ADDERRORBRANCHES(P, Ψ)
6: B := PATHCRAWLER(P', *Context*)
7: **return** B
8: **end if**

ADDERRORBRANCHES(P, Ψ)

1: **for all** $i \in I$ **do**
2: **if** $\Psi_i \equiv$ *false* **then**
3: $\alpha'_i := \alpha_i$ /* no alarm for α_i */
4: **else**
5: $\alpha'_i :=$ if(Ψ_i) storeBugAndExit(); else α_i
6: **end if**
7: **end for**
8: **return** $P' = \{\alpha'_i \mid i \in I\}$

3 Experiments

In this section, we compare our combined method with static analysis and with two test generation techniques used independently. The first testing technique is running PathCrawler with various strategies but without any information on threatening statements. The second one, denoted *all-threats*, runs PathCrawler in alarm-guided mode like SANTE, but for the exhaustive list of alarms for all potentially threatening statements (i.e. with potential risk of a run-time error).

We use five examples shown in Fig. 1 extracted from real-life software where bugs were previously detected. All bugs are out-of-range indices or invalid pointers. Examples 1–4 come from Verisec C analysis benchmark [9], example 5 from [10]. The columns of Fig. 1 respectively present the example number, its origin, the name of the analyzed function, the size of each example in lines of code, the total number of potential threats, the number of known bugs among them, and the results of value analysis. Fig. 2 compares SANTE to other test generation techniques. Its columns respectively show the example number, the PathCrawler strategy (test selection criterion) and the results for each method. The column 'safe' provides the number of threats proven unreachable by value analysis or by exhaustive all-paths testing when it terminates. The column 'unknown' provides the number of remaining unconfirmed alarms (relevant for value analysis, PathCrawler all-threats and SANTE). We also present, when relevant, the number of bugs detected, the number of treated paths, and full process duration. The strategy k-path is given for the minimal k allowing to detect all bugs in SANTE. Experiments were conducted on an Intel Duo 1.66 GHz notebook with 1 GB of RAM with a 30 min timeout.

SANTE vs. static analysis. Fig. 1 shows that in most cases static analysis alone reduces the number of potential threats and proves that some of them are safe, but still generates many alarms. We see in Fig. 2 that SANTE confirms some alarms as real bugs, provides a test case activating each bug and leaves less unknown alarms.

SANTE vs. PathCrawler alone. SANTE detects more bugs than PathCrawler alone, and treats additional paths arising from error branches with reasonable extra time (Fig. 2, see for instance Ex. 3).

SANTE vs. PathCrawler all-threats. Alarm-guided test generation in SANTE only treats the alarms raised by value analysis while all-threats dully considers all potential threats. Thus test generation in SANTE considers less paths, detects the same

	origin	function name	size (loc)	all threats	known bugs	value analysis		
						safe	unknown	time
1	Apache	escape_absolute_uri (simplified)	33	8	1	4	**4**	1s
2	Apache	escape_absolute_uri (full)	97	16	1	11	**5**	1s
3	Spam Assassin	message_write	55	17	2	2	**15**	1s
4	Apache	get_tag	165	12	3	0	**12**	2s
5	QuickSort	partition	50	8	1	4	**4**	1s

Fig. 1. Examples and static analysis results

	strategy	PathCrawler alone			PathCrawler all-threats					SANTE				
		bugs	paths	time	safe	unknown	bugs	paths	time	safe	unknown	bugs	paths	time
1	all-paths	**0**	2164	14s	7	**0**	1	3602	**22s**	7	**0**	**1**	2454	14s
	3-path	**0**	30	<1s	0	**7**	1	71	**<1s**	4	**3**	**1**	45	<1s
2	all-paths	**0**	2023	10s	15	**0**	1	3876	**20s**	15	**0**	**1**	2694	13s
	10-path	**0**	232	1s	0	**15**	1	417	**1s**	11	**4**	**1**	325	1s
3	all-paths	**0**	31917	311s	time / space out					15	**0**	**2**	37967	523s
	3-path	**0**	12446	120s	0	**15**	2	30977	**558s**	2	**13**	**2**	18874	215s
4	all-paths	time / space out			time / space out					time / space out				
	2-path	**1**	26595	663s	0	**9**	3	36690	**870s**	0	**9**	**3**	36690	872s
5	all-paths	**1**	5986	33s	7	**0**	1	15216	**116s**	7	**0**	**1**	11893	72s
	2-path	**1**	569	5s	0	**7**	1	4509	**25s**	4	**3**	**1**	3319	18s

Fig. 2. Experimental results for two test generation techniques and our combined method

number of bugs in less time and leaves less unknown alarms. It terminates in some cases where all-threats spaces/times out (Ex. 3). In the worst case, when static analysis can't filter any threat, SANTE can take as much time as all-threats (cf Ex. 4, 2-path).

Additional application of program slicing before alarm-guided test generation didn't show obvious gain here, because these examples were already simplified.

4 Related Work and Conclusion

Closely related work. Many static and dynamic analysis tools are well known and widely used in practice. Recently, several papers presented combinations of dynamic and static methods for program verification, e.g. [1–6]. Daikon [4] uses dynamic analysis to detect likely invariants. [5] compares two combined tools for Java: Check 'n' Crash and DSD-Crasher. Our all-threats method is similar to [11], called *active property checking* in [6]. Synergy/Dash [3] and BLAST [2] combine testing and partition refinement for property checking. The idea of combining static analysis and testing for debugging was mentioned in [6] but was not implemented and evaluated.

Conclusion. We have presented our ongoing research on a new method combining static analysis and structural testing, as well as experimental results showing that this method is more precise than a static analyzer and more efficient in terms of time and number of detected bugs than a concolic structural testing tool alone or guided by the exhaustive list of alarms for all potentially threatening statements. Static analysis alone will in general just generate alarms (some of which may be false alarms), whereas our method allows to confirm some alarms as real bugs and provides a test case activating each bug. This is done automatically, avoiding, at least for confirmed

alarms, time-consuming alarm analysis by the validation engineer, requiring significant expertise, experience and deep knowledge of source code. Stand-alone test generation, when it is not guided by generated alarms for some statements, does not detect as many bugs as our combined method. When guided by the exhaustive list of alarms for all potentially threatening statements (not filtered by static analysis), test generation usually has to examine more infeasible paths and takes more time than our combined method (or even times/spaces out). In all cases, our method outperforms each technique used independently. Since complete all-paths testing is unrealistic for industrial software, it is also encouraging to see that realistic partial criteria (e.g. k-path) are very efficient in SANTE method. We expect that other testing techniques will also gain from the use of static analysis as concolic testing evaluated here.

Future work includes continuing research to eliminate unconfirmed alarms, to better support other alarm types (e.g. invalid pointers) and to integrate program slicing; experimenting with other coverage criteria (e.g. all-branches) and with breadth-first search; extending the SANTE implementation and comparing it with other tools and on more benchmarks.

Acknowledgments. The authors thank the members of the PathCrawler and Frama-C teams for providing the tools and support. Special thanks to Loic Correnson, Bernard Botella and Bruno Marre for their helpful advice and fruitful suggestions.

References

1. Pasareanu, C., Pelanek, R., Visser, W.: Concrete model checking with abstract matching and refinement. In: CAV. (2005) 52–66
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker BLAST: Applications to software engineering. Int. J. Softw. Tools Technol. Transfer **9**(5-6) (2007) 505–525
3. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: FSE. (2006) 117–127
4. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3) (2007) 35–45
5. Smaragdakis, Y., Csallner, C.: Combining static and dynamic reasoning for bug detection. In: TAP. (2007) 1–16
6. Godefroid, P., Levin, M.Y., Molnar, D.A.: Active property checking. In: EMSOFT. (2008) 207–216
7. Frama-C: Framework for static analysis of C programs (2007-2010) http://www.frama-c.com/.
8. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST. (2009) 70–78
9. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE. (2007) 389–392
10. Ball, T.: A theory of predicate-complete test coverage and generation. In: FMCO. (2004) 1–22
11. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In: CCS. (2006) 322–335