VERMEER: A Tool for Tracing and Explaining Faulty C Programs

Daniel Schwartz-Narbonne*, Chanseok Oh*, Martin Schäf†, and Thomas Wies*
*New York University
†SRI International

***Abstract*—We present VERMEER, a new automated debugging tool for C. VERMEER combines two functionalities: (1) a dynamic tracer that produces a linearized trace from a faulty C program and a given test input; and (2) a static analyzer that explains why the trace fails. The tool works in phases that simplify the input program to a linear trace, which is then analyzed using an automated theorem prover to produce the explanation. The output of each phase is a valid C program. VERMEER is able to produce useful explanations of non trivial traces for real C programs within a few seconds.**

The tool demo can be found at http://youtu.be/E5lKHNJVerU.

I. INTRODUCTION

Although sophisticated tools have emerged in the past years that automate many tasks in the software development process, debugging still remains a time consuming and predominantly manual activity. To improve automated debugging support, we present VERMEER, a tool that automatically traces individual faulty executions of C programs and produces an explanation of the faulty behavior. The tool is organized in several analysis passes each of which produces an executable C program. This maximizes the ease of interaction with the programmer and yields high interoperability with other analysis tools.

We explain the key functionality provided by VERMEER through an example. A programmer faced with a bug has two basic questions: what happened, and why? Consider the following C code:

```
1 int a = 100, b=2;
2 while(a > 0){
3   a = abs(-1 * (a-1));
4   b++;
5 }
6 assert(b < 4);
```

Our tool VERMEER helps programmers answer the question of "what happened" by providing them with a C file which represents the exact trace of instructions executed by the program. In fact, the tool provides several views on this trace at different levels of abstraction.

The first view is simply the *linear trace* of the executed instructions where calls to library functions are kept as explicit calls. The linear trace for the above program is:

```
1 ...
2 tmp5 = (a) - (1);
3 tmp6 = (-1) * (tmp5);
4 a = abs(tmp6);
5 b = (b) + (1); // Assigned: 4
6 ... 1700 more lines
7 assert(b < 4LL);
```

Currently, VERMEER requires a program specification in the form of a violated assertion. Implicit bugs can be represented by adding an assertion, e.g., that a pointer was non-null or an array index was in-bounds.

The next view is a *concrete trace* where all calls of library functions have been replaced by their effect on the program state:[1]

```
1 ...
2 tmp5 = (a) - (1);
3 tmp6 = (-1) * (tmp5);
4 // Call: a = abs(tmp6);
5 if (tmp6 == -98){ a = 98; }
6 b = (b) + (1); // Assigned: 4
7 ... 1700 more lines
8 assert(b < 4LL);
```

Notice how the call to `abs` has been replaced by a conditional assignment that tracks the dependency on the concrete output of `abs` on its concrete input.

The programmer still wonders: why did the trace fail? To help answer this question, VERMEER implements an error explanation algorithm. The generated explanation for the above trace is

```
1 b = 2LL;                  //line: 1
2 //(<= 0 (+ b (- 2 ) ) )
3 b = b + 1LL;              //line: 4
4 //(<= 0 (+ b (- 3 ) ) )
5 b = b + 1LL;              //line: 4
6 //(<= 0 (+ b (- 4 ) ) )
7 assert(b < 4LL);          //line: 6
```

The lines represent instructions, while the comments represent *error invariants* [1], [2]. These are program assertions that are (1) true at that program point (2) sufficient to guarantee that the error will be reached. In this case, the explanation tells us that after two executions of the loop, $b \geq 4$ becomes an invariant for the remaining instructions of the trace. VERMEER therefore decides that all remaining 1700 lines of the trace are irrelevant for understanding the faulty behavior. That is, the explanation tells us that the original program failed because we executed the loop at least two times. Any execution with two or more iterations of the loop will reach the error.

VERMEER is distributed under a BSD license. The source code distribution and all benchmark programs are available on the tool website [3].

[1] VERMEER keeps track of the line numbers in the original program that each instruction in the trace emanated from. We do not show this information here for conciseness.

This work was supported in part by NSF grant CCS-1350574.

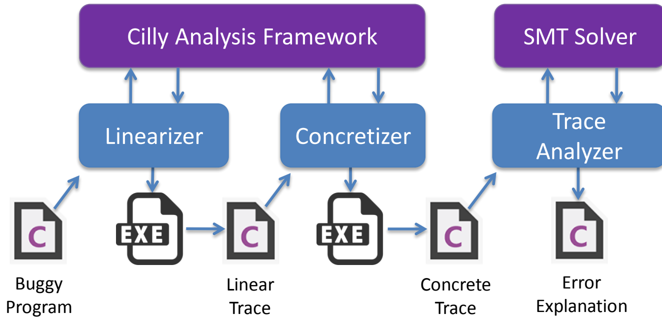

Fig. 1. VERMEER architecture

II. ARCHITECTURE

VERMEER is structured as a series of analysis passes, each of which takes a C program as input, and outputs an executable C program, as shown in Figure 1. Since each pass performs a simple transformation, they are easy to test and debug. In addition, the "linear" and "concrete" traces form useful abstractions for interfacing with other verification tools.

Linearizer. The first pass takes the original faulty C program, and uses a custom CILLY [4] analysis plugin to produce an annotated executable. When this new program is executed, it emits a loop-free, branch-free C program which contains every instruction executed by the original program (it also inlines all function calls for which the original source code was available). We refer to the output of this process as a "linear trace".

Concretizer. The linear trace can then be processed by a second custom CILLY analysis plugin to produce a second annotated executable which, when executed, emits a C program in which all interactions with the environment are represented by their concrete effect on local state. In particular, all external function calls including IO are represented by a combination of their return value and side effects and all heap memory is transformed into local variables. We refer to the output of this process as a "concrete trace".

Trace Analyzer. The trace analyzer inputs a concrete trace, and applies our fault abstraction algorithm described in [1], [2]. This algorithm uses a combination of static slicing and an SMT solver to generate an annotated trace which contains only the statements necessary for the fault *along with an explanation for why those statements mattered.*

III. IMPLEMENTATION

A. Logging format

Input. VERMEER is capable of handling almost all C constructs, including memory allocation, function pointers, structures, arrays, library functions and IO. Operation with side-effects such as copying structures and IO are handled by using wrapper functions, which we have been adding to VERMEER on an as-needed basis when we encounter a benchmark that requires them. Floating point values are difficult to represent exactly, and the trace may suffer from round-off-error if floating point is used. Inline assembly could be handled using the techniques from [5]. Support for concurrency is ongoing work.

Output. Traces generated by VERMEER are valid C programs. VERMEER traces can be compiled and executed using standard compilers. They can be used as input to any C analysis tool. As an added benefit, since VERMEER traces are standard C programs, they are human readable.

B. Linearizer

The basic design of the linearizer and concretizer is simple: we use a CILLY C program analysis plugin (719 lines of OCAML code) to create an annotated program in which every statement is preceded by a call to a logging function which records the program location and program statement to a log. Currently, this function directly prints C instructions as strings to a file, but it would be possible to use a binary encoding if required.

Normalizing the AST. One challenge for analyzing C code is that a single C line of code may contain several logical operations, which can make it hard to analyze. (e.g. consider `for(i=0;j>=0&&i<=5;i++,j++)` or `a[i]->b = c[j]->d`). We take advantage of the "simplify" option in CILLY to transform the program into simple three-address code before processing it. This also reduces the work of further analysis passes, because it ensures that operations in the trace will appear in a canonical form.

Inlining functions. VERMEER inlines function calls whenever possible. Since different functions may have local variables with conflicting names, we prefix each variable with the function name. We handle recursive function calls by maintaining a "scope index" which counts the current function stack depth. The variable `x` used in function `f` with stack depth 3 would therefore be `f___3___x`. This encoding has an added benefit of making the current call context explicit in the trace.

Post-processing. If the program under test has a crash, then the log will end at the statement that triggered the crash, which may not constitute a valid C program. We use a PERL script to post-process the trace into a valid program.

C. Concretizer

Like the linearizer, the concretizer is a CILLY plugin (793 OCAML LoC) which compiles a linear trace into a program which, when run, will emit a concrete trace.

Concrete memory accesses. One of the biggest challenges for static analysis is tracking heap memory accesses. Because we are dealing with a simple linear trace, we can determine at runtime the location of every memory access, and replace a heap memory access to location `0x2048` with the corresponding operation to local variable `mem_0x2048`. For efficiency, we currently assume that the program was type-safe, and do not support type-unsafe operations such as bit-casting.

Concrete function calls. Calls to functions that cannot be inlined, such as library functions, are represented by recording their effect into the trace. For side-effect-free calls, this simply means recording the function's return value. VERMEER has a mechanism for wrapping functions with side-effects so that

TABLE I

SIZE AND EXECUTION TIME OF TRACES.

Program	Prog. LoC	Execution Time (ms)			Linear Tr. Size		Conc. Tr. Size		# Instructions (# Original Lines)			
		Prog.	Linear	Conc.	LoC	KB	LoC	KB	Sliced	Expl.	Fast Abs.	Full Abs.
simple	13	48	46	51	413	17	441	24	506 (203)	203 (3)	4 (3)	5 (3)
tcas	137	43	47	50	342	12	367	17	9 (5)	6 (4)	6 (4)	6 (4)
schedule2	301	48	53	58	6917	335	7076	389	389 (228)	24 (4)	24 (4)	24 (4)
replace	513	40	53	51	4329	187	3454	190	269 (184)	48 (25)	48 (25)	48 (25)
gzip	4956	40	46	47	2903	152	3305	169	353 (152)	82 (31)	74 (30)	74 (30)
sed	9293	44	54	46	6930	348	3054	167	116 (57)	11 (8)	11 (8)	11 (8)
grep	9521	46	108	135	36162	2162	40389	2479	3988 (1451)	727 (710)	39 (22)	40 (23)
MiniSat	1100	48	114	118	49793	2568	38057	2474	4511 (2100)	128 (75)	126 (73)	126 (73)

they will emit their side-effects onto the trace. These wrappers have been written on an as-needed basis and currently cover a sufficient portion of `stdlib` to enable our benchmarks.

In order to capture the functional dependence of calls on their output, VERMEER guards its concrete function representation inside a conditional which represents the known inputs to the function, as shown in the concrete trace in § I.

D. Trace Analyzer

The trace analyzer consists of three separate sub-phases.

Slicing. First, we reduce the concrete trace using a simple data-flow aware (but data-value insensitive) static slicing algorithm, implemented as an OCAML plugin to CILLY (228 LoC). Next, we convert the concrete trace to SSA form (207 lines of OCAML code) in order to enable the data-value sensitive analysis sub-phases.

Explanation. Next, we use an SMT solver to compute the unsat core of the SSA trace. This reduces the trace to the relevant statements. The reduced trace is then passed as another SMT query, which computes Craig interpolants to obtain the error invariants that explain the causes of the fault.

Abstraction. Finally, we use the computed error invariants to abstract the trace to a more compact error explanation. The abstraction eliminates those statements in the reduced trace that have no effect modulo the computed error invariants. For example, in the trace that we considered in § I, it is this step that eliminates the 1700 instructions that come from the redundant loop iterations. We have implemented two versions of this algorithm. The first version, which we call *fast abstraction*, simply propagates the error invariants from the previous step across the trace to identify redundant statements. This is the original algorithm described in [1]. The second algorithm, which we call *full abstraction*, computes new interpolants after each successful propagation. This ensures that the error invariants in the resulting explanation are inductive with respect to the remaining non-redundant statements. Full abstraction typically produces explanations that are easier to understand. However, this analysis is also more costly because it involves repeated calls to the interpolating SMT solver. Interpolation involves proof-producing SMT queries, which are slower than regular non proof-producing queries.

In our implementation, we are using the SMT solver SMTINTERPOL for all interpolation queries and Z3 for all remaining queries. However, it would be easy to replace these by other solvers and we are planning to add support for more solvers in the future. Currently, we present invarients in STMLIB format. Work to convert invariants to simplified C is ongoing.

IV. EVALUATION

Benchmarks. We evaluated VERMEER using a set of real world benchmarks drawn from the Software-artifact Infrastructure Repository (SIR) [6]. SIR benchmarks come prepackaged with the inputs necessary to trigger the bug. Since VERMEER currently focuses on data-flow analysis, we preferentially chose examples which had data-flow related bugs. In addition, we have added two new benchmarks. MINISAT is an SAT solver written in C [7], which we seeded with a fault and the input required to reach the bug. We have also included the "simple" example from our tool demo. The types of faults range from memory access violation to semantic errors. All experiments were run on an Intel Core i7-4770 CPU @ 3.40GHz with 16GB of RAM.

As described in § III-C, we concretized the environment for our benchmark programs using wrappers for `stdlib` functions. In addition, a few programs required some manual modification to work in our tool. MINISAT and schedule2 use floating point, which is currently not supported by SMT-INTERPOL, so we modified those programs to use a fixed point representation. grep uses struct- and union-copying, which we converted to field-wise copying. tcas, schedule2 and replace were written for Solaris, and required minor modifications to compile on Linux.

Results. Table I and Table II summarize the results of our evaluation. The first column of Table I shows the size of the programs in terms of the number of lines. The next three columns show execution time of the faulty program, the binary that generates its linearized trace, and the binary that generates the concretized trace, respectively, all of which follow the same execution path to reach the seeded fault. The next four columns show the size of the intermediate error traces. The final columns show how long each "analyzed" trace is. The first set of numbers in each column represents the number of canonical instructions that have been expanded from actual code, while the number in parenthesis represents the number of lines of code.[2] For example, we reduced an actual error trace

[2]Since one line of original C code typically expands to multiple instructions, we count consecutive instructions with a same line number tag as one line.

TABLE II
TIME SPENT IN EACH STEP OF THE TRACE ANALYSIS (SEC.)

Program	Static Slicing	SSA Conv.	SMT Time (#SMT Calls) Expl.	Fast Abs.	Full Abs.
simple	0.06	0.08	0.65	0.94 (206)	0.48 (38)
tcas	0.07	0.06	0.22	0.22 (12)	0.28 (36)
schedule2	0.19	0.07	0.32	0.34 (31)	0.69 (128)
replace	0.12	0.07	0.42	0.53 (88)	1.26 (314)
gzip	0.10	0.06	0.67	0.90 (148)	2.26 (498)
sed	0.09	0.06	0.22	0.26 (22)	0.36 (71)
grep	9.95	0.47	5.99	9.98 (1444)	171.60 (268)
MiniSat	5.15	0.54	1.34	1.79 (256)	3.95 (888)

of the faulty program grep (40,389 LoC) to a surprisingly small sequence of 40 canonical instructions that concisely explains the reason for the error, which maps back to 23 lines of original code. As discussed in § III-D, each of these instructions is associated with an invariant that describes why it led to the bug. Table II shows time required to run the analyzer in seconds, broken down into three sequential phases: simple static-slicing, SSA conversion, and the computation of the abstract trace using SMT. We can see that the both the "explain" and "fast abstraction" algorithms are extremely fast, completing in seconds (often less than a single second). As expected "full abstraction" is somewhat slower, because it makes more calls to the computation intensive interpolation procedure. We are exploring a number of techniques including the use of other SMT solvers, incremental solving, and trace windowing to improve the performance of this analysis.

Our results show that VERMEER scales to work in real-world benchmarks giving succinct explanations to errors within seconds.

V. RELATED WORK AND CONCLUSION

Lately several automated debugging tools have emerged that use theorem provers to generate fault explanations from proofs of unsatisfiability. One of the first tools of this kind was BUGASSIST [8], which uses a MAX-SAT solver to identify a small subset of statements in a C that are likely candidates for the root cause of an observed bug. Similar to VERMEER, whose underlying algorithm is described in [1], [2], the root cause candidates are computed from an unsatisfiable formula. Our approach uses interpolation instead of MAX-SAT and focuses on fault explanation rather than localization. The explanations generated by VERMEER contain error invariants that further highlight the relevant variables and the relation between them. Error invariants also often yield more compact explanations than a pure MAX-SAT analysis. BUGASSIST does not use dynamic tracing to reduce and simplify the formula, which limits its scalability. However, our dynamic tracer could be combined with BUGASSIST. A hybrid approach that combines our interpolation-based analysis with a MAX-SAT analysis is described in [9].

Another related approach for explaining errors found by static analysis tools is described in [10]. This approach uses abduction, which can be considered the logical dual of interpolation. As in the case of BUGASSIST, it could benefit from the dynamic tracing implemented in VERMEER.

Fault localization with theorem provers has been successfully applied in the hardware domain. For instance, [11] and [12] use proofs to localize hardware design errors. VERMEER targets software, which brings a different type of challenges.

There are other approaches that have successfully combined static and dynamic analysis for fault localization and explanation in different contexts. The authors of [13] use unsat cores to improve the quality of a testing based fault localization. F3 [14] uses a combination of static and dynamic analysis to localize faults by finding closely related passing and failing test cases. Chandra et al. [15] propose a technique called angelic debugging which identifies expressions that are likely to cause an execution to fail. The advantage of using VERMEER in comparison to these approaches is that VERMEER only requires a single failing execution which makes it easier and cheaper to use. However, taking into account successful computations in the fault analysis can be beneficial in certain cases.

In our future work, we will extend VERMEER with the flow-sensitive fault localization technique described in [16], so that our trace explanation algorithm can also track branching conditions. Moreover, to further increase the scalability of the tool, we will implement the windowing technique described in [2]. This way, the SMT solver will only operate on a bounded window of the trace at any time.

REFERENCES

[1] E. Ermis, M. Schäf, and T. Wies, "Error invariants," in *FM*, 2012.
[2] C. Oh, M. Schäf, D. Schwartz-Narbonne, and T. Wies, "Concolic fault abstraction," in *SCAM*, 2014.
[3] "Vermeer tool web page," http://cs.nyu.edu/wies/software/vermeer.
[4] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C programs," in *CC*, 2002.
[5] S. Maus, M. Moskal, and W. Schulte, "Vx86: x86 assembler simulated in C powered by automated theorem proving," in *AMAST*, 2008.
[6] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *EMSE*, 2005.
[7] N. Eén and N. Sörensson, "An extensible sat-solver," in *SAT*, 2003.
[8] M. Jose and R. Majumdar, "Bug-Assist: Assisting fault localization in ANSI-C programs," in *CAV*, 2011.
[9] V. Murali, N. Sinha, E. Torlak, and S. Chandra, "What gives? a hybrid algorithm for error trace explanation," in *VSTTE*, 2014.
[10] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *PLDI*, 2012.
[11] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," in *CAV*, 2009.
[12] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *GLSVLSI*, 2008.
[13] D. Gopinath, R. N. Zaeem, and S. Khurshid, "Improving the effectiveness of spectra-based fault localization using specifications," in *ASE*, 2012.
[14] W. Jin and A. Orso, "F3: Fault localization for field failures," in *ISSTA*, 2013.
[15] S. Chandra, E. Torlak, S. Barman, and R. Bodik, "Angelic debugging," in *ICSE*, 2011.
[16] J. Christ, E. Ermis, M. Schäf, and T. Wies, "Flow-sensitive fault localization," in *VMCAI*, 2013.