Loopfrog: A Static Analyzer for ANSI-C Programs

Daniel Kroening*, Natasha Sharygina[†], Stefano Tonetta[‡], Aliaksei Tsitovich[†] and Christoph M. Wintersteiger[§]
* *Computing Laboratory, Oxford University, Oxford, UK*
[†] *University of Lugano, Switzerland*
[‡] *Fondazione Bruno Kessler, Trento, Italy*
[§] *Computer Systems Institute, ETH Zurich, Switzerland*

Abstract—**Practical software verification is dominated by two major classes of techniques. The first is model checking, which provides total precision, but suffers from the state space explosion problem. The second is abstract interpretation, which is usually much less demanding, but often returns a high number of false positives. We present LOOPFROG, a static analyzer that combines the best of both worlds: the precision of model checking and the performance of abstract interpretation. In contrast to traditional static analyzers, it also provides 'leaping' counterexamples to aid in the diagnosis of errors.**

I. INTRODUCTION

Loops in programs are the Achilles' heel of static analysis. A sound analysis of all program paths through loops requires either an explicit unwinding or an over-approximation (of an invariant) of the loop. Unwinding is computationally too expensive for many real programs, and the computation of sufficiently strong invariants is an art.

LOOPFROG is an automatic bug-finding tool for ANSI-C programs. Unlike traditional program approximation approaches (e.g. abstract interpretation [1]) it does not employ iterative fixpoint computation, instead it uses a *summarization* algorithm [2] that non-iteratively computes *symbolic abstract transformers* with respect to a set of abstract domains. Summaries are shorter, loop-free program fragments, which are used to substitute the original loops to obtain a conservative abstraction of the program. LOOPFROG computes abstract transformers starting from the inner-most loop. It obtains a loop invariant by checking if the constraints defined by a chosen abstract domain are preserved by the loop. These checks are performed by means of calls to a SAT-based decision procedure, which allows us to check (possibly infinite) sets of states with one query. Thus, unlike other approaches [3], [4], LOOPFROG is not restricted to finite-height domains.

When all the loops are summarized, the resulting over-approximation of the program is handed to a bit-precise model checker [5]. Due to the simplicity of the summarized program, the state space explosion problem of the model checker is avoided. A theoretical treatment of our algorithms has been published before in [2]; in this paper we present their implementation.

Supported by the Swiss National Science Foundation under grant no. 200021-111687.

Related Work: Polyspace and Astrée are two well-known static analysis tools that implement traditional abstract interpretation. They are optimized for embedded control software. Both rely on iterative fixpoint computation, and may require many iterations, depending on the complexity of the program. Other tools include Calysto [6] and Saturn [7]; both are more precise and scalable than traditional static analyzers, and, just like LOOPFROG, implement bit-precise reasoning for all ANSI-C constructs. While these tools also make use of loop and function summaries with respect to an abstract domain, they trade soundness for scalability, e.g., by unwinding loops a constant number of times. Another tool for ANSI-C program analysis is CBMC [8] — a bounded model checker that implements bit-precise reasoning without abstraction. It unwinds loops up to a pre-set bound and checks for assertion violations by means of a SAT-based decision procedure. In practice, it often needs to unwind loops up to a very high bound. For non-terminating loops, it may not terminate at all. LOOPFROG is partly based on CBMC's symbolic execution engine, but does not unwind loops.

II. LOOPFROG

The theoretical concept of symbolic abstract transformers is implemented and put to use by our tool as outlined in Fig. 1. As input, LOOPFROG receives a model file, extracted from software sources by GOTO-CC[1]. This model extractor features full ANSI-C support and simplifies verification of software projects that require complex build systems. It mimics the behaviour of the compiler, and thus 'compiles' a model file using the original settings and options. Switching from compilation mode to verification mode is thus achieved by changing a single option in the build system. As suggested by Fig. 1, all other steps are fully automated.

The resulting model contains a control flow graph and a symbol table, i.e., it is an intermediate representation of the original program in a single file. For calls to system library functions, abstractions containing assertions (pre-condition checks) and assumptions (post-conditions) are inserted. Note that the model also contains the properties to be checked in the form of assertions (calls to the ASSERT function).

[1]http://www.cprover.org/goto-cc/

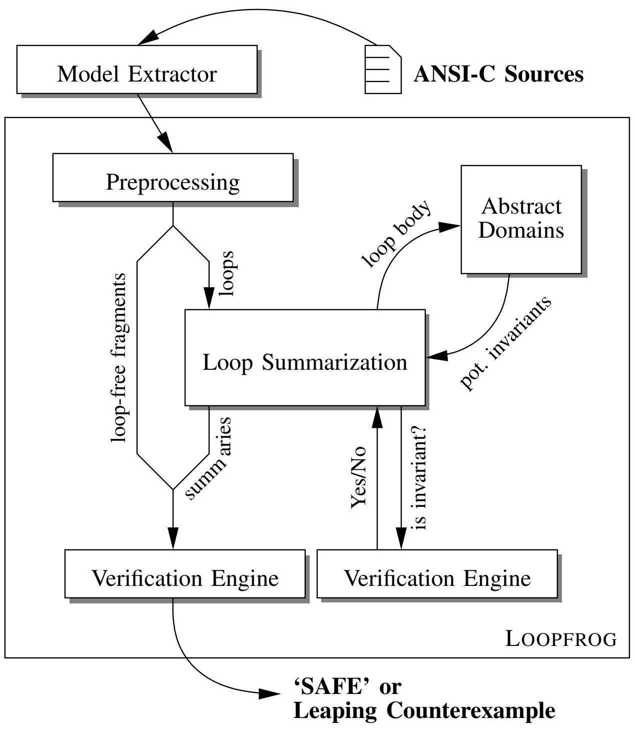

Figure 1. Architecture of LOOPFROG

Preprocessing: The instrumented model is what is passed to the first stage of LOOPFROG. In this preprocessing stage, the model is adjusted in various ways to increase performance and precision. First, irreducible control flow graphs are rewritten according to an algorithm due to Ashcroft and Manna [9]. Like in a compiler, small functions are inlined. This increases the model size, but also improves the precision of subsequent analyses. After this, LOOPFROG runs a field-sensitive pointer analysis. The information obtained through this is used to insert assertions over pointers, and to eliminate pointer variables in the program where possible. On request, LOOPFROG automatically adds assertions to verify the correctness of pointer operations, array bounds, and arithmetic overflows.

Loop Summarization: Once the preprocessing is finished, LOOPFROG starts to replace loops in the program with *summaries*. These are shorter, loop-less program fragments that over-approximate the original program behaviour. To accomplish this soundly [2], all loops are replaced with a *loop-less* piece of code that 'havocs' the program state, i.e., it resets all variables that may be changed by the loop to unknown values. Additionally, a copy of the loop body is kept, such that assertions within the loop are preserved.

While this is already enough to prove some simple properties, much higher precision is required for more complex ones. As indicated in Fig. 1, LOOPFROG makes use of predefined abstract domains to achieve this. Every loop body of the model is passed to a set of abstract domains, through each of which a set of potential invariants of the loop is derived (heuristically). In the current version, LOOPFROG comes with a set of abstract domains that are specialized to buffer-related properties, in order to demonstrate the benefits of our approach on buffer-overflow benchmarks. For example, there are constraints that are able to express that an index into a buffer is always within the bounds of the buffer (Table I gives more examples).

All potential invariants obtained from abstract domains always constitute an abstract (post-)state of the loop body, which may or may not be correct in the original program. To ascertain that a potential invariant is an actual invariant, LOOPFROG makes use of a verification engine. In the current version, the symbolic execution engine of CBMC [8] is used. This engine allows for bit-precise, symbolic reasoning without abstraction. In our context, it always gives a definite answer, since only loop-less program fragments are passed to it. It is only necessary to construct an intermediate program that assumes the potential invariant to be true, executes the loop body once and then checks if the potential invariant still holds. If the verification engine returns a counterexample, we know that the potential invariant does not hold; in the contrary case it must be a true invariant and it is subsequently added to the loop summary, since even after the program state is havoced, the invariant still holds. LOOPFROG starts this process from the innermost loop, and thus there is never an intermediate program that contains a loop. In case of nested loops, the inner loop is replaced with a summary, before the outer loop is analyzed. Due to this and the shortness of the fragments checked (only the loop body), small formulas are given to the verification engine and an answer is obtained quickly.

Verification: The result, after all loops have been summarized, is a loop-less abstraction of the input program. This abstract model is then handed to another verification engine. Again, the verification time is much lower than that of the original program, due to the model not containing any loops. As indicated by Fig. 1, the verification engine used to check the assertions in the abstract model, may be different from the one used to check potential invariants. In LOOPFROG, we choose to use the same engine (CBMC). We do so for two reasons: 1) it is very efficient and 2) it returns counterexamples in case of assertion violations. The

Table I
EXAMPLE DOMAINS FOR BUFFER-OVERFLOW ANALYSIS.

#	Constraint	Meaning
1	ZT_s	String s is zero-terminated
2	$L_s < B_s$	Length of s (L_s) is less than the size of the allocated buffer (B_s)
3	$0 \leq i \leq L_s$	Bounds on integer variables i (i is non-negative, i is bounded by buffer size, etc.) k is an arbitrary integer constant.
4	$0 \leq i$	
5	$0 \leq i < B_s$	
6	$0 \leq i < B_s - k$	
7	$0 < offset(p) \leq B_s$	Pointer offset bounds
8	$valid(p)$	Pointer p points to a valid object

Table II
A COMPARISON BETWEEN LOOPFROG AND AN INTERVAL DOMAIN: THE COLUMN LABELLED 'TOTAL' INDICATES THE NUMBER OF PROPERTIES IN THE PROGRAM, AND 'FAILED' SHOWS HOW MANY OF THE PROPERTIES WERE REPORTED AS FAILING; 'RATIO' IS FAILED/TOTAL.

			LOOPFROG		Interval Domain	
Suite	Benchmark	Total	Failed	Ratio	Failed	Ratio
bchunk	bchunk	96	8	0.08	34	0.35
freecell-solver	make-gnome-freecell-board	145	40	0.28	140	0.97
freecell-solver	make-microsoft-freecell-board	61	30	0.49	58	0.95
freecell-solver	pi-make-microsoft-freecell-board	65	30	0.46	58	0.89
gnupg	make-dns-cert	19	5	0.26	19	1.00
gnupg	mk-tdata	6	0	0.00	6	1.00
inn	encode	42	11	0.26	38	0.90
inn	ninpaths	56	19	0.34	42	0.75
ncompress	compress	204	38	0.19	167	0.82
texinfo	makedoc	83	46	0.55	83	1.00
wu-ftpd	ckconfig	1	1	1.00	1	1.00
wu-ftpd	ftpcount	61	7	0.11	47	0.77
wu-ftpd	ftpshut	63	13	0.21	63	1.00
wu-ftpd	ftpwho	61	7	0.11	47	0.77

counterexamples in our setting are abstract, i.e., they do not contain any information about the program states visited in loops and are therefore called *leaping* counterexamples.

III. EXPECTED BENEFITS

Through the GOTO-CC model extractor, LOOPFROG allows users to quickly switch from a compilation setting to a verification setting. Compared to other static analyzers, the 'verification overhead' is kept minimal.

Traditional static analyzers are often impractical due to their high false positive rates, while model checking is often too demanding in terms of runtime. LOOPFROG provides a solution to those problems by combining the two techniques into one scalable, sound *and* precise analysis. The abstract domains employed to achieve the precision are usually simple and may be implemented with few lines of code.[2]

On top of this, LOOPFROG also provides leaping counterexamples if it finds a bug – a definite advantage over many other static analyzers based on traditional techniques, and an invaluable aid in understanding why a bug is reported.

IV. EVALUATION AND AVAILABILITY

LOOPFROG was previously evaluated on a wide range of benchmarks ranging from regression tests to independently composed benchmarks suites [10] and large-scale open-source software like GNUPG, INN, and WU-FTPD. It was shown that it outperforms many other static analysis tools in terms of performance and false positive rate [2].

To highlight the applicability of LOOPFROG to large-scale software and to demonstrate its main advantage, we present a new comparative evaluation against a simple interval domain, which tracks the bounds of buffer index variables, an often employed static analysis. For this experiment, LOOPFROG was configured to use only two abstract domains, which capture the fact that an index is within the buffer bounds (#4 and #5 in Table I). As apparent from Table II, the performance of LOOPFROG in this experiment is far superior to that of the simple static analysis.

[2]Future versions of LOOPFROG will support custom domains.

We analyze a single benchmark in detail, in order to explain the data delivered by the tool: The `ncompress` program (version 4.2.4) contains about 2.2K lines of code (which translates to 963 instructions in the model file) and 12 loops. During preprocessing, LOOPFROG detected 204 potential buffer overflows and inserted an assertion for each of them in the model. Loop summarization took 14.4 seconds. During this time, 67 potential invariants were created and 17 of them were confirmed as actual invariants. The overall analysis took 668 seconds.[3] Finally, 166 assertions hold and 38 are reported as failing (while producing leaping counterexamples for each violation).

To evaluate scalability, we applied other verification techniques to this example. CBMC [8] tries to unwind all the loops, but fails, reaching the 2GB memory limit. The same behaviour is observed using SATABS [4], where the underlying model checker (SMV) hits the memory limit.

LOOPFROG, extensive experimental data, and all our benchmark and test files are available on-line for experimentation by other researchers.[4]

REFERENCES

[1] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*. ACM, 1977.

[2] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger, "Loop summarization using abstract transformers," in *ATVA*, ser. LNCS 5311. Springer, 2008.

[3] T. W. Reps, S. Sagiv, and G. Yorsh, "Symbolic Implementation of the Best Transformer," in *VMCAI*, ser. LNCS 2937. Springer, 2004.

[4] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-based predicate abstraction for ANSI-C," in *TACAS*, ser. LNCS 3440. Springer, 2005.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[6] D. Babić and A. J. Hu, "Calysto: Scalable and Precise Extended Static Checking," in *ICSE*. ACM, 2008.

[7] A. Aiken, S. Bugrara, I. Dillig, T. Dillig, B. Hackett, and P. Hawkins, "An overview of the Saturn project," in *PASTE*. ACM, 2007.

[8] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *TACAS*, ser. LNCS 2988. Springer, 2004.

[9] E. Ashcroft and Z. Manna, "The translation of 'go to' programs to 'while' programs," in *Classics in software engineering*. Yourdon Press, 1979.

[10] K. Ku, T. Hart, M. Chechik, and D. Lie, "A buffer overflow benchmark for software model checkers," in *ASE*. ACM, 2007.

[3]On an Intel Xeon 3.0GHz, 4GB RAM.
[4]http://www.verify.inf.unisi.ch/loopfrog/