-OVERIFY: Optimizing Programs for Fast Verification

Jonas Wagner Volodymyr Kuznetsov George Candea
School of Computer and Communication Sciences
École polytechnique fédérale de Lausanne (EPFL), Switzerland

Abstract

Developers rely on automated testing and verification tools to gain confidence in their software. The input to such tools is often generated by compilers that have been designed to generate code that runs fast, not code that can be verified easily and quickly. This makes the verification tool's task unnecessarily hard.

We propose that compilers support a new kind of switch, -OVERIFY, that generates code optimized for the needs of verification tools. We implemented this idea for one class of verification (symbolic execution) and found that, when run on the Coreutils suite of UNIX utilities, it reduces verification time by up to 95×.

1 Introduction

Automated program verification tools are essential to writing good quality software. They find bugs and are crucial in safety-critical workflows. For example, for a bounded input size, symbolic execution engines like KLEE [4] and Cloud9 [3] can verify the absence of bugs in systems software like Coreutils, Apache, and Memcached; the SAGE [2] tool found many security vulnerabilities in Microsoft Office and Windows. Unfortunately, verification tools are not used widely in practice, because they are considered too slow and imprecise.

Many verification tools, including the ones mentioned above, verify *compiled* programs. The observation underlying the work presented in this paper is that verification complexity could be significantly reduced if only a program was compiled specifically for verification. Building on this point, we argue that compilers should have an -OVERIFY option, which optimizes for fast and precise verification of the compiled code.

This paper shows how the requirements of verification differ from those of fast execution on a CPU. We identify program transformations that reduce verification complexity, and others that increase it. We also present ideas for new transformations that, although not commonly performed by today's compilers, would be of great benefit to verification tools. In this way, we hope to make verification an integral part of the software build chain.

-OVERIFY is a step toward enabling wide-spread use of verification tools: it gets developers the verification results faster, is easy to integrate into existing build chains and, since it uses time-tested compiler technology, it generates reliable verification results.

```
int wc(unsigned char *str, int any) {
  int res = 0;
  int new_word = 1;

  for (unsigned char *p = str; *p; ++p) {
    if (isspace(*p) ||
        (any && !isalpha(*p))) {
      new_word = 1;
    } else {
      if (new_word) {
        ++res;
        new_word = 0;
      }
    }
  }

  return res;
}
```

Listing 1: Count words in a string; they are separated by whitespace or, if any≠0, by non-alphabetic characters.

Motivating example: The example in Listing 1, a function that counts words in a string, illustrates the effect of compiler transformations on program analysis.

This simple function is challenging to analyze for several reasons: It contains an input-dependent loop whose number of iterations cannot be statically predicted. Inside the loop, the control flow branches on the type of the current input character. This leads to an explosion in the number of possible paths—there are $O(3^{\text{length}(str)})$ paths through this function. Finally, wc calls the external library functions isspace and isalpha, whose implementations further complicate the analysis of wc.

We exhaustively tested all paths through wc for inputs up to 10 characters long using KLEE [4], and Table 1 shows the results for four different compiler settings. Aggressive optimizations can dramatically reduce verification time. At level -O2, the reduction comes mostly from algebraic simplifications and removal of redundant operations, which lead to a reduced number of instructions to interpret in KLEE. The number of explored paths remains the same as for -O0, indicating that -O2 does not fundamentally change the program's structure.

At level -O3, the compiler unswitches the loop, i.e., the loop-invariant condition any $\neq 0$ is moved out of the loop, and simplified copies of the loop body are emitted for any $= 0$ and any $\neq 0$, respectively. This enables algebraic simplification of the branch condition, which in turn reduces the number of paths to $O(2^{\text{length}(str)})$ and thus reduces verification time. The price for this reduction is an increase in the size of the compiled program, due to the duplicated loop body.

Optimization	*-O0*	*-O2*	*-O3*	-OVERIFY
t_{verify} [msec]	13,126	8,079	**736**	**49**
t_{compile} [msec]	38	42	43	44
t_{run} [msec]	3,318	704	694	1,827
# instructions	896,853	480,229	37,829	312
# paths	30,537	30,537	2,045	11

Table 1: Using symbolic execution to exhaustively explore all paths in the code of Listing 1 for strings of up to 10 characters: time to verify (t_{verify}), time to compile (t_{compile}), and time to run on a text with 10^8 words (t_{run}).

Compiling the program using -OVERIFY goes beyond the optimizations performed by -O3, and completely removes all branches from the loop (see Listing 2). This reduces the number of paths to $O(\text{length}(str))$, and symbolic execution time decreases by 15×.

A traditional compiler would not perform this optimization, because the cost of executing a branch on a CPU is so small that it is cheaper to perform a branch that skips some instructions than to unconditionally execute the entire loop body. Indeed, when actually executed, the branch-free version takes 2.5× as long as the -O3 branching version (Table 1). This illustrates the conflicting requirements of fast execution and fast verification.

```
int sp = isspace(*p) != 0;
sp |= (any != 0) & !isalpha(*p);
res += ~sp & new_word;
new_word = sp;
```

Listing 2: Branch-free version of wc's loop body.

2 The Case for -OVERIFY

One usually thinks about a compiler as a tool that translates programs into the code that a CPU can execute. While doing this translation, the compiler transforms programs to make them execute as quickly as possible, taking into account CPU properties like instruction costs, pipeline architecture, or caching behavior.

Today, the compiler output is consumed in a number of ways beyond its original purpose of executing on a CPU. Here, we target verification tools that consume the compiler's output, either in its final binary form or in an intermediate representation, such as LLVM bitcode [13]. What such tools expect from a compiler is different from, and sometimes contradictory to, the requirements imposed by execution performance considerations.

2.1 Compiling for Program Verification

The *time* to verify a program is dominated by the number of branches it has, the overall number of loop iterations, memory accesses, and various arithmetic artifacts.

The *precision* of the analysis can also depend on the program structure, e.g., on the number of control-flow points or the kinds of operations the program performs.

Verification tools can often exploit high-level knowledge of the program, like information about types of variables or about the program's use of the standard library.

Compilers affect all these aspects, and can make programs more verification-friendly using techniques described in Section 3. Does this mean that it becomes possible to verify systems that previously were out of reach for verification tools? For simple verification tools that employ coarse-grained abstractions, the answer is yes: compiler transformations can increase their precision and allow them to prove more facts about a program. A more complex and precise tool will be able to verify any semantically equivalent version of a program. Yet how much information it has to infer, and hence its running time, depends on the way a program is presented to it. This explains the speedups we report in Section 4.

2.2 -OVERIFY Belongs in the Compiler

Why perform verification optimizations inside compilers? Why not let every verification tool transform source code using its own specialized transformations?

Making -OVERIFY part of a compiler has three key advantages: First, compilers are in a unique position in the build chain. They have access to the most high-level form of a program (its source code) and control all steps until the program is in the low-level intermediate form that is to be analyzed. Second, the program needs to be compiled anyway, and so -OVERIFY gains access to a wealth of information, like call graphs and alias analysis results, at no additional cost. Third, with -OVERIFY, verification tools need not be aware of complex build chains, and need not re-implement common transformations.

We do not advocate a monolithic approach in which compilers and verifiers are tightly coupled. Instead, we want to put to more effective use the services that a compiler offers, and reduce the amount of information lost during compilation. This opens new questions for the systems community, spanning the entire process from software development to verification: At what levels should verification be performed? What abstractions should compilers export, so that clients can express their needs and customize the build process? What data should be preserved across the different program transformation steps, and in what format? How can we guarantee the correctness of program transformations?

2.3 Using -OVERIFY in Practice

Developers usually create different build configurations for software systems. During development, a program is compiled with debug information, assertions, and possibly reduced optimization to aid testing and debugging. At release time, the program is compiled with the highest

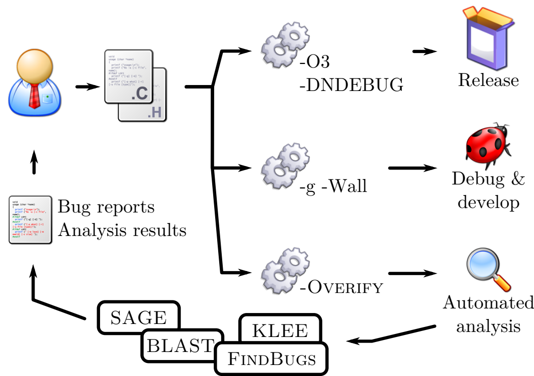

Figure 3: Adding -OVERIFY to an existing build chain, to enable fast automated program analysis and testing.

optimization level. Our proposed -OVERIFY option adds a third build configuration, aimed at automated testing and verification. This process is illustrated in Figure 3.

An -OVERIFY-enabled compiler can be directly leveraged by a number of program analysis tools. We built a prototype that can generate special binaries optimized for analysis using the S^{2}E system [5] or SAGE tool [11], which perform symbolic execution on x86 binaries. Alternatively, it can generate LLVM IR bitcode optimized for analysis by tools like KLEE and its descendants [4, 15, 3]. We expect the ideas in this paper to apply broadly to many other tools, such as FindBugs for Java [10], or Microsoft Pex [18] for .NET.

-OVERIFY makes it possible to use verification-specific optimizations with minimal changes to a build chain, lowering the entry barrier to the use of verification tools available today. We envision future project creation wizards and build systems creating -OVERIFY configurations by default, thus encouraging wide adoption of powerful verification tools, which in turn can help build better software and improve developer productivity.

Verifying a verification-optimized version of the program and then shipping a performance-optimized version means that end-users get not exactly what was tested and verified. However, -OVERIFY relies on time-tested compiler transformations that are anyway used (perhaps differently) in the rest of the build chain, and this offers confidence in the equivalence of the verified and shipped versions. Programs that contain undefined behavior face the largest risk. For example, a function that returns the address of a local variable might behave differently when the function is inlined. Compilers can often, but not always, detect such cases and warn the developers.

3 A Design for -OVERIFY

While a compiled program must be semantically equivalent to its source code representation, a compiler still has a lot of room for optimizations. In Table 2 we show the various options a typical compiler could offer today to -OVERIFY, as well as options it *does not* offer.

-OVERIFY modifies the compilation process in four complementary ways: (1) it selects a set of compiler passes suitable for verification tools, and it inhibits compiler passes that would increase verification complexity; (2) it adjusts cost values and parameters (such as the maximum size of a function to inline) to optimize compilation for fast verification, not fast execution; (3) it causes more metadata to be preserved in the program; and (4) it links the program with a specialized version of the C standard library optimized for verification.

We now give a few examples of the large body of possible program transformations and illustrate the effect they have on verification complexity.

Simplifying control flow. Program verification often becomes drastically easier if the program's control flow is simplified by a compiler. An optimization called *jump threading* checks whether a conditional branch jumps to a location where another condition is subsumed by the first one; if yes, the first branch is redirected correspondingly, turning two jumps into one. Another example is *loop unswitching*, as seen in Section 1.

These are especially important for verification tools that reason about multiple execution paths through a program (either individually or grouped in some way). For such tools, the complexity of verifying a program depends on the number of possible execution paths through it, which in turn grows exponentially with the number of conditional branches and the number of possible loop iterations. As a result, branches and loops have a much higher cost for verification than for normal execution.

Control flow can be further simplified by transforming conditionally executed side-effect-free statements into speculative branch-free versions. This is a standard transformation done when saving one branch instruction outweighs the cost of speculation (e.g., GCC converts `if (test) x = 0;` into `x &= −(test == 0);`). When using -OVERIFY, this simplification is pursued more aggressively, because the cost of a branch is higher.

Restructuring the program. Data-flow based verification tools reason about programs at the granularity of program locations. If multiple paths lead to one location, merging the path information can lose precision. Unswitching a loop can improve precision, because such information merging now occurs only once, after the loop, instead of after every iteration. Similarly, *loop unrolling* and *function inlining* increase the number of program locations and can thus increase precision.

Instruction simplification. Standard simplifications, such as *copy propagation* and *constant folding*, are good for execution speed, but can be even better for verification. Consider a tool that reasons about value

Transformation	Verification	Execution	Available
Constant propagation/folding, arithmetic simplifications	$+$	$+$	✓
Remove/split memory accesses	$+$	$+$	✓
Simplify control flow: jump threading, loop unswitching	$+$	$+/-$	✓
Restructure the program: function inlining, loop unrolling	$+/-$	$+/-$	✓
Improve cache behavior, register allocation, instruction scheduling	$-$	$+$	✓
Program annotations: types, alias information, loop trip counts	$+$	$-$	few
Generate runtime checks	$+$	$-$	some

Table 2: Compiler optimizations and their impact (positive $+$, or negative $-$) on *Verification* time and/or *Execution* time. The last column shows to what degree the optimizations are readily available in today's compilers.

ranges for variables: When encountering code such as `x = input(); y = x; x -= y;` the tool might think that x could have any value. Yet standard simplifications can turn this code into the equivalent but easier version `input(); x = 0`.

Memory accesses complicate the data-flow graph of a program, requiring verification tools to analyze which accesses may correspond to the same memory location vs. which cannot (alias analysis). The complexity of this analysis typically grows exponentially with the number of related memory accesses.

A compiler can easily help by converting values that reside in memory to register values, and by splitting large objects into independent smaller objects, thereby reducing the opportunities for memory access aliasing.

Program annotations. The output of today's compilers does not preserve all information present in the source code of the program, such as high level types or the separation of a program into modules. Compilers also do not keep information computed during compilation, such as alias information, variable ranges, loop invariants, or trip counts. This information however is priceless for verification tools, and could be easily preserved in the form of program metadata. Some of these are available today, e.g., the Clang compiler [6] can annotate memory accesses with types.

Runtime checks. Recent versions of Clang and GCC can emit run-time checks for various forms of illegal behavior, transforming these various failures into run-time crashes. This makes verification simpler, as tools now only need to check for one type of failure (i.e., crashes).

CPU-specific optimizations. To generate code that executes fast, compilers must optimize for the target CPU's cache structure and pipeline: keep loops small (to avoid instruction cache misses), pad objects (to keep them aligned in memory), and reorder instructions (to reduce pipeline stalls and to improve branch prediction).

All these issues are irrelevant to many classes of verification tools, and some of these optimizations can even slow down verification. Thus, they are omitted under the -OVERIFY switch. This offers the further benefit of considerably more freedom in generating code.

Library-level changes. For programs that use the C/C++ standard library, the analysis effort depends significantly on the complexity of library functions. This is why some tools, such as KLEE and KLOVER [15], ship with a custom version of the C/C++ standard library.

As part of -OVERIFY, we are currently developing a version of libC that is tailored to the needs of program analysis in general, and thus reusable for many tools. This library provides versions of the standard functions designed to minimize the analysis costs outlined in Section 2.1. These simplifications entail high-level reasoning and semantic understanding of the code that is beyond what modern compilers can do automatically.

Functions in this C library contain run-time checks to verify their preconditions. Such checks are often absent or disabled in production code. For testing and verification, these checks improve the tools' ability to find bugs, and they also lead to better error reports, because bugs are found closer to their root cause.

Does -OVERIFY generalize? One could argue that each verification tool requires its own specialized version of -OVERIFY. Yet the *idea* behind -OVERIFY generalizes, and we expect that developers of verification tools can readily decide—intuitively and based on our examples—which optimizations would be advantageous. Compilers can help them by providing access to built-in heuristics (e.g., to decide when a function should be inlined), as well as heuristics specialized for -OVERIFY (e.g., heuristics for estimating when speculative execution would reduce analysis time [12]).

4 Prototype and Early Results

We implemented a prototype, called -OSYMBEX, that makes verification easier for symbolic execution tools like SAGE, KLEE, and others. These tools analyze programs one path at a time. They treat program inputs as *symbolic*, i.e., they assume inputs can have any value (up to a bounded size). As it interprets the program, such

Optimization	*-O0*	*-O3*	-OSYMBEX
# functions inlined	0	7,746	16,505
# loops unswitched	0	377	3,022
# loops unrolled	0	1,615	3,299
# branches converted	0	959	5,405

Table 3: Compiling Coreutils with different options.

a tool keeps track of all the symbolic expressions computed by the program. At conditional branches, a constraint solver is invoked to check whether the symbolic condition could be true, false, or both. In the latter case, the tool explores both paths independently, adding the branch condition (or its negation, respectively) as a constraint on the inputs for the current path.

The performance of symbolic execution tools is determined by the number of paths to explore and by the complexity of input-dependent branch conditions. Our prototype -OSYMBEX reduces both, thereby improving the performance of symbolic execution tools without requiring the tools themselves to be modified.

We built -OSYMBEX on top of the LLVM compiler infrastructure. Compared to -O3, -OSYMBEX: (1) considers the cost of a branch to be higher than on a CPU, to avoid branches through speculative execution and loop unswitching; (2) removes loops from the program whenever possible, even if this increases the program size; and (3) aggressively inlines functions in order to benefit from simplifications due to function specialization. Table 3 shows how these changes affect the number of program transformations performed by the compiler.

We evaluated -OSYMBEX on real systems code: we ran it on the Coreutils 6.10 suite of UNIX utilities, in essence repeating the case-study from [4].

Figure 4 shows the effect -OSYMBEX has on the verification of Coreutils. For each of the 93 tested programs, we measured how long it takes to compile and analyze all paths in KLEE, using 2 to 10 bytes of symbolic input. We did this with -O0, -O3, and -OSYMBEX, respectively. We kept all experiments where KLEE terminates within one hour on at least one of the three versions.

On average, -OSYMBEX reduces overall compilation and analysis time by 58% compared to -O3, and by 63% over -O0. The maximum benefit is a 95× reduction in total time (right side of Figure 4). The verification of 6 programs runs out of time with -O3 (and 11 with -O0), but completes with -OSYMBEX. In a few cases, -O3 outperforms -OSYMBEX, because it takes longer to compile with -OSYMBEX than -O3; this effect vanishes in longer experiments. We verified that indeed all bugs discovered by KLEE with -O0 and -O3 are also found with -OSYMBEX.

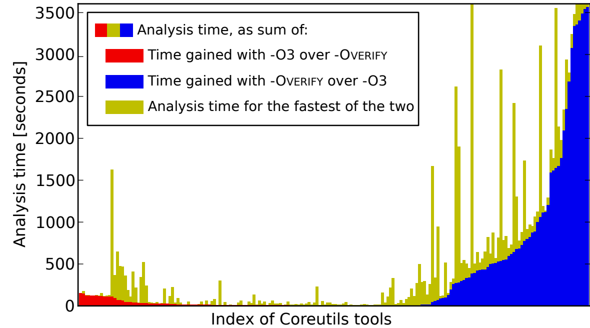

Figure 4: Time to compile and test Coreutils; each bar represents one experiment. Blue (sorted on the right) represents time gained by using -OVERIFY over -O3; red (on the left) shows when -O3 is faster than -OVERIFY; yellow shows the time of whichever one is fastest.

5 Related Work

The KLEE [4] symbolic execution tool is one of the success stories of applying symbolic execution to systems software. This success was partly due to using an adapted C library based on uClibc and a symbolic file system model. Simplifications of the model and library made symbolic execution tractable. Some successors of KLEE continued using specific optimizations: The authors of KLEE-FP report that aggressive speculative execution improved symbolic execution performance [8]. A number of C++ library functions have been re-written for the KLOVER tool in order to achieve the same goal [15].

Several verification tools use CIL [17] to preprocess C code and thus benefit from some compiler transformations. Some recent tools are more closely integrated into compilers, such as Microsoft's Verifying C Compiler [7], the LLVM-based tools UFO [1] and LLBMC [16], and the GCC plugin Predator [9]. -OVERIFY shares with them the leveraging of the compiler, but remains modular and applicable to a variety of verification methods.

Programs can be compiled to Boogie [14], a specialized intermediate language for verification. This makes them even easier to verify than compiling with -OVERIFY, but they lose their ability to be executed.

6 Conclusion

Starting from the observation that compiling a program to run on a CPU has different requirements than compiling the program for verification, we propose adding an -OVERIFY switch to compilers. This switch applies optimizations specifically geared toward fast verification. With a preliminary prototype we were able to reduce verification time by up to 95× and by 58% on average.

References

[1] A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Intl. Conf. on Computer Aided Verification*, 2012.

[2] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. Technical Report MSR-TR-2012-55, Microsoft Research, 2012.

[3] S. Bucur, V. Ureche, C. Zamfir, and G. Candea. Parallel symbolic execution for automated real-world software testing. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[4] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Sys. Design and Implem.*, 2008.

[5] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[6] The Clang compiler. http://clang.llvm.org/.

[7] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, 2009.

[8] P. Collingbourne, C. Cadar, and P. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *ACM EuroSys European Conf. on Computer Systems*, 2011.

[9] K. Dudka, P. Müller, P. Peringer, and T. Vojnar. Predator: a verification tool for programs with dynamic linked data structures. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2012.

[10] Findbugs – find bugs in Java programs. http://findbugs.sourceforge.net/.

[11] P. Godefroid, M. Y. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp.*, 2008.

[12] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implem.*, 2012.

[13] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *Intl. Symp. on Code Generation and Optimization*, 2004.

[14] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, 2010.

[15] G. Li, I. Ghosh, and S. Rajan. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *Intl. Conf. on Computer Aided Verification*, 2011.

[16] F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. *Verified Software: Theories, Tools, Experiments*, 2012.

[17] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Intl. Conf. on Compiler Construction*, 2002.

[18] N. Tillmann and J. De Halleux. Pex – white box test generation for .NET. *Tests and Proofs*, 2008.