*Моделирование и анализ информационных систем.* Т. 22, № 6 (2015), с. 763–772
*Modeling and Analysis of Information Systems.* Vol. 22, No 6 (2015), pp. 763–772



**DOI:** 10.18255/1818-1015-2015-6-763-772

UDC 004.054+004.4'23

# Fast and Safe Concrete Code Execution for Reinforcing Static Analysis and Verification

Belyaev M., Itsykson V.

*Received September 15, 2015*

The problem of improving precision of static analysis and verification techniques for C is hard due to simplification assumptions these techniques make about the code model. We present a novel approach to improving precision by executing the code model in a controlled environment that captures program errors and contract violations in a memory and time efficient way. We implemented this approach as an executor module `Tassadar` as a part of bounded model checker `Borealis`. We tested `Tassadar` on two test sets, showing that its impact on performance of `Borealis` is minimal.

The article is published in the authors' wording.

**Keywords:** concrete interpretation, symbolic execution, static code analysis, analysis precision

**For citation:** Belyaev M., Itsykson V., "Fast and Safe Concrete Code Execution for Reinforcing Static Analysis and Verification", *Modeling and Analysis of Information Systems*, **22**:6 (2015), 763–772.

**On the authors:**

Belyaev Mikhail, orcid.org/0000-0003-1260-9211, assistant,
Peter the Great St. Petersburg Polytechnic University,
Polytechnicheskaya street, 21, Saint-Petersburg, 194021, Russia
e-mail: belyaev@kspt.icc.spbstu.ru

Itsykson Vladimir, orcid.org/0000-0003-0276-4517, PhD,
Peter the Great St. Petersburg Polytechnic University,
Polytechnicheskaya street, 21, Saint-Petersburg, 194021, Russia
e-mail: vlad@icc.spbstu.ru

# Introduction

Static analysis and verification of programs written in an unsafe language like C is hard. Most of the problems researchers face in these areas are either NP-hard or undecidable due to inherent properties of a Turing-complete language and the presence of unsafe memory operations. Another difficulty comes from the fact that the analysis that can be used in an interprocedural environment must be aware of both internal and external functions to provide a good approximation of program behaviour.

The goal of this work is to provide a safe and fast way to reduce the number of false positives in results produced by static code analysis of C. We mainly focus on logic-based analysis techniques (such as bounded model checking [5]) that allow to find non-functional program defects (i.e. null pointer deferences, buffer overflows, division by zero, etc.) and code contract violations. We provide a way to reduce false positives in these cases in the form of a concrete code execution environment that borrows some ideas from symbolic execution and is safe, robust and resource-efficient.

This technique has been implemented in a prototype executor called `Tassadar` as a module of a bounded model checker `Borealis` [1] and was tested using `Borealis` testbench.

The rest of this paper is structured as follows. The problem statement is given in section 1. Section 2 is dedicated to describing the execution environment itself. The implementation details are given in section 3. Section 4 includes our evaluation results. Related work and summary of alternate approaches are given in section 5.

# 1. Problem Statement

Most approaches to finding program defects and/or checking source contracts have limitations that result in lowering code model conformance as a trade-off between the quality of analysis and resource consumption. The most basic of these limitations are as follows.

1. Limiting the number of loop iterations and recursive function calls;
2. Simplified view on memory and pointer aliasing;
3. Using summaries and/or contracts to model function calls;
4. Replacing stdlib/system/external function calls with annotations or approximated models.

For the rest of this paper we will use the term *analysis* (or *the* analysis) to refer to the static analysis or verification technique that is being augmented by our approach and assume that the aim of this analysis is to either detect program defects or contract violations, referred to as *errors*.

There are two basic measures for analysis quality: *precision* and *recall* [6]. Precision and recall are dual properties that cannot usually be achieved together at the same time: any improvement to recall usually raises the number of false positives as well, lowering precision, and any change that increases precision usually lowers the number of true positives, having a negative impact on recall. **The idea of this work** is to improve analysis precision by filtering out false positives *after* the analysis has finished, thus

having no impact on recall. Improving precision is very important for real-life usage of defect detection tools as false positives make analysis results noisy and less usable.

We do this by using a technique that is based on directly executing all program procedures in a checked and analysis-aware environment. The set of properties this process must satisfy is as follows.

- Avoid infinite and lengthy execution;
- Minimize resource consumption to the lowest level possible;
- Avoid as many of model inaccuracies imposed by the analysis as possible;
- Capture all analysis-supported errors.

We must also keep notice that some values (e.g., a call to `rand()` that results in a defect only for some return values) must be handled in an analysis-conformant way, returning the same value that was inferred by the analysis if possible. This is also true for initial function arguments and global variables for a concrete function execution. Without this, it would be impossible to assess whether the analysis-provided result was correct.

# 2. Code Execution in a Controlled Environment

This section details our approach, including dealing with standard register operations, memory simulation and calling external libraries. We also provide some details on how we deal with code contracts.

## 2.1. Modeling register operations

In this paper we assume that the code model conforms to the static single assignment (SSA [8]) form. This is a widely used code model (e.g., used by GCC [14] and LLVM [13] infrastructures). This provides us with ability to work on already optimized code for everything besides memory operations. Implementing numeric operations is pretty straightforward using biggest numeric types available. All the work with structures/unions is done via memory.

Calling internal functions is done through modeling call stack and current instruction pointer. As all the values in SSA form are assigned only once, we can store their values in a flat value table. We also keep track of stack-allocated pointers for deallocation. In order to conform to the variability of results provided by analysis techniques, some values in the code should be treated as unknown, or *symbolic*. The biggest difference between our approach and symbolic execution is that we always try to get concrete values for all these symbols using a special facility called *the arbiter*.

Arbiter is a function from symbols to real values that has external information about the code that the executor have no access to. When checking analysis-based counterexamples, it is the arbiter that provides the connection between the executor and the analysis. Each query for a value that cannot be directly computed results in a mapping of the corresponding variable from the counterexample. However, the arbiter interface is independent from the analysis itself, providing a point of extension.

A special case of symbolic values are pointers. All the symbolic pointers are modeled using a special non-null value that is known both to arbiter and memory model. Dereferencing this value effectively queries the arbiter instead of the memory.

Currently we do not support writing values to symbolic pointers and this presents a problem when dealing with structures passed by pointer to top-level functions, but this problem can be dealt with by reconstructing the storage graph from the counterexample, which is possible but complex and not yet implemented in our prototype.

## 2.2. Modeling memory operations

*Segment tree* (ST) is a well-known data structure first introduced by [3]. It is essentially a balanced tree with leaves representing elements and inner nodes representing ranges, the root of the tree being the whole range and every child representing a range that is one half of its parent's. This data structure can be used to efficiently perform a number of tasks, like the minimum range query or any other range query based on an associative binary operation. Changing a leaf value and recalculating the whole tree is a $O(log_2(N))$ operation for a range of size $N$.

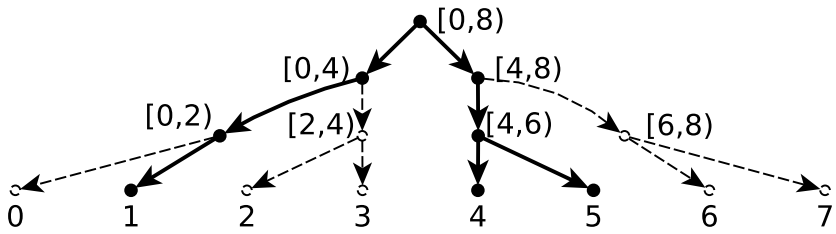


Fig. 1. Implicit segment tree

This structure is very inefficient memory-wise. This can be countered by using an *implicit* version of segment tree, where tree nodes are constructed during writes only. Figure 1 shows this difference in detail: an ST needs 15 cells for 8 elements, while an implicit ST (shown in bold) only 7 cells for 3 writes. This change imposes no penalty on lookup complexity. Memory consumption becomes $O(Klog_2(N))$ where $K$ is the number of write queries. The difference can be dramatic for big arrays. E.g., for a range of size $2^{31}$ building an ST will require 4294967294 cells, while for the implicit version it depends on the number of writes, which is much smaller. Another optimization is to reduce height by storing fixed size flat arrays in leaves, which reduces the height by $log_2(P)$ but adds $O(P)$ to the complexity of each write.

As mentioned earier, an implicit ST can be efficiently used to represent big amounts of data without consuming too many resources. In order to provide all the desired memory properties for our executor, like handling buffer overflows, allocations, deallocations and so on, we need to store this cumulative information in the nodes themselves in a manner similar to the one used when solving range queries.

Each node in the tree contains a tuple $\{Q, S, F, Sz, D\}$. Memory status $Q$ (`Alloca`, `Malloc` or `Unknown`) designates whether this exact node was used as the root of an allocated memory region with size $Sz$. Memory state $S$ (`Memset`, `Uninitialized` or `Unknown`) is used to represent zero-initialized, uninitialized or `memset` region of memory,

enabling shortcut reads for regions that are allocated, but never written to. The byte $F$ represents the byte value for `memset`.

Each ST leaf contains a pointer to buffer containing the memory chunk and does not contain child pointers. The usual memory access pattern in C implies that the size of chunks stored in leaves should be at least as big as structure padding and preferably a multiple of it. With exception to direct memory access functions and unpadded structures, this will ensure that every read or write accesses one chunk only.

Summing up, every node in the tree represents a memory range of size $2^N K$ where $K$ is the size of leaf memory chunk. To allocate a new memory range we need to find a node representing a big enough range and set $Q$ and $S$ accordingly. No nodes below this level are created until written to.

To perform a write we descend down the tree until we reach the desired chunk, splitting the data source array and creating new nodes if needed. An important thing is keeping track of the value of $S$. For each new child created, we create its sibling, copy $S$ and $F$ from parent and reset $S$ in parent to unknown. A read is the same, but does not create new nodes and merges the data array on the way back up. If we encounter a node with $S = \texttt{Memset}$, the returned value is set to $F$.

Other primitive operations are `memset` (different to write in that it assigns $S$ and $F$ for appropriate ranges) and `memchr` (similar to read except that it traverses the tree horisontally) because there is no way to do them efficiently through reads and writes.

On each tree traversal a *single allocated* node on the way down is searched for and bound-checked. Any situation when this is violated is an error.

## 2.3. Modeling global state

Global variables and constants have a much simpler structure. All values in this space have known size and need no additional checks for initialization [10]. In order to keep track of global variables, we use a standard binary search tree of pointers to lazily-allocated buffers. In the future, it might be more convenient to use a unified structure for both local and global data. Pointers to labels and functions are stored separately in a structure associating them with real pointers. Doing arithmetics on these pointers is an error.

The only query needed for global data is to be able to find the base pointer for a value. This is done using a standard binary search. Local and global memories are distinguished by the ranges of corresponding pointers.

## 2.4. Handling external functions

We do support all the functions of the standard C library and the most common POSIX functions. The system-based side effects, like networking, files, etc. are not simulated, but the values produced are sensible for their counterparts, including error situations. Functions without side effects (for example, the builtin math library) are implemented directly.

Functions directly operating the memory are implemented on top of primitive reads, writes, `memset` and `memchr`. Another function that is subject to become primitive is

`memcpy`, which is rather inefficient at the moment and making it efficient is a subject of further research.

Another important thing is support for external libraries, which at the moment can only be implemented as a part of the executor itself. Providing an external interface to implement these is also a subject of future research.

### 2.5. Checking code contracts

Most programming tools that support contracts either specify them inside the language (implementing these is no different from other functions) or outside, be it comments, annotations or external data files. These definitions must be parsed separately and are usually provided as their AST. As these contracts are (usually) not allowed to have any side effects, thus executing them is straightforward using the AST itself.

Some systems (notably the ACSL annotation language used in Frama-C [2]) provide a framework for specifying contract in a more logic-oriented way, supporting quantifiers over logic formulae, providing user-created theorems and other constructs that are more typical for logic programming languages rather than functional or imperative ones and cannot be in any way *executed* in the sense we put to this term. The reason for this is that "executing" a quantified formula effectively means checking it for every possible value of all bound variables, which violates our goals and is impossible in the general case. However, in some simple cases these can be reduced to non-quantified variants by means of skolemization (or hebrandization) of these formulae. In other cases, the set of values for each bound variable is known at runtime and they can be checked iteratively. At the moment, such contracts are ignored. Approximating checks with these contracts in a reasonable way is a subject for future research.

## 3. Implementation details

We have implemented our approach in a module called `Tassadar` in a bounded model checker tool `Borealis` [1] using the infrastructure borrowed from LLVM interpreter library. `Borealis` is based on LLVM IR and supports several ways of specifying code contracts. The LLVM interpreter is a simple instruction interpreter that operates on LLVM IR instructions and in order to implement the checks needed by our tool the whole memory model, external function calls and some parts of regular instruction handling had to be written from scratch. We also implemented executing and checking `Borealis` contract specifications.

`Borealis` is based on using an SMT-solver for checking desired properties. Every found defect or contract violation provides a counterexample — a set of values that could lead to an erroneous situation at run-time. We re-package the values from this counterexample to a hashtable and use that as the basis for our analysis-driven arbiter. The whole solution is implemented in C++ and packaged as a set of LLVM passes that are run by `Borealis`.

# 4. Evaluation

We evaluated our approach and `Tassadar` on two test case sets bundled with `Borealis` which are based on NECLA [11] and SV-COMP [4] test case packs, which test both defect detection and contract violation detection properties of the checker. This includes 84 testcases with over 40 defects/violations, about 30% of which are false positives in the first set and 20 test cases with around 12 defects/violations with 30% false positive ratio in the second set. The main difference is that the second set contains much more complex programs with heavy memory usage. `Tassadar` was required to check every positive detected by `Borealis`. In total it has identified 16 false positives in both test sets and no false negatives.

Table 1. Test run results summary

| | Time+, s | Time−, s | Time% | Mem+, | Mem−, | Mem% | FPs |
|---|---|---|---|---|---|---|---|
| Set 1 | 19.017 | 18.976 | 0.2% | 71366 | 70189 | 17% | 11 |
| Set 2 | 107.844 | 107.678 | 0.1% | 173553 | 173697 | 0.08% | 5 |

The test runs were performed on a AMD Phenom(tm) II X4 machine with 4 cores and 8 Gbs of RAM. The test bench was analyzed 10 times with `Tassadar` and without it. The results of the runs are summarized in table 1. Time+ and Time− is time spend with executor and without it, Mem+ and Mem− is the top memory consumption for both. The "%" columns show the approximate percentage of corresponding resource spent by `Tassadar`. The main goal of this work is to provide a result checker that has minimal impact compared to the run time of the analysis. As one can see from the table, this impact does never exceed 1 percent of total run time.

# 5. Related work

The techniques based on augmenting analysis techniques through with "simulation" of produced counterexamples is well-known as the basis for abstraction refinement-based approaches, e.g. for traditional model checking [7]. It was later extended by [12] to actually *execute* the code to provide information suitable for refinement. This and later workings (for example, [9]) propose implementing abstraction refinement through code-to-code instrumentation, compiling and running instrumented code using traditional means.

This is quite similar to our approach, albeit for a different purpose. A similar technique could be implemented for our problem, but using instrumented code execution limits the ability to reason about inner state of the program. None of these papers actually give a description of data structures used to capture, store and query memory range information, but it is hinted at that they use a simple linear container of ranges in addition to the memory itself, which has linear time and memory complexity over the number of ranges. The program itself should have means of communication (through replacing different external functions) with the checker during runtime. This approach is also not capable of capturing some buffer overflows and illegal pointer dereference when illegal pointers accidentally point to legal data.

[9] provide evaluation results for their approach and state the total execution overhead

as 1% to 12% of all runtime, which is much worse than our approach gets, but they are not directly comparable due to different analysis techniques used.

# Conclusion

This paper is focused around building an efficient executor of C code for reinforcing results of static analysis using dynamic result checks. We build our approach around creating an interpreter that operates on SSA-based compiler model that allows us to use the compiler-based standard optimizations and provides a controlled environment that is able to capture and check all the analyzed properties at the same time.

We have built a prototype implementation on top of the LLVM compiler system and `Borealis` bounded model checker that has proven to be very resource-efficient with comparison to the checker itself and, at the same time, being able to correctly identify a portion of false positives.

In the future we plan to apply this executor to improving the analysis quality further, building a CEGAR-like refinement loop on top of it, support the things we do not currently support (fully rebuild top function arguments passed by pointer, use an external language for external functions, explore the possibilities of checking quantifier-based and intrinsic contracts, etc.) and do a more thorough comparison with other similar tools.

We also plan to research the possibilities of applying the existing implementation to different areas it could be used in, like checking and reducing tests produced by test generation (of which `Borealis` has limited support [**?**]), checking results for other kinds of analysis, explore the possibility for dynamic analysis in general to check arbitrary code properties during execution. It is also possible to try building a memory-efficient time-travelling interpreter on top of `Tassadar` as the data structure used to model the memory is persistent and implementing versioning on top of it is pretty straightforward.

# References

[1] M. Akhin, M. Belyaev, V. Itsykson, "Software defect detection by combining bounded model checking and approximations of functions", *Automatic Control and Computer Sciences*, **48**:7 (2014), 389–397.

[2] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, V. Prevosto, 2008, ACSL: ANSI/ISO C Specification Language. Preliminary Design, version 1.4., Preliminary.

[3] J. L. Bentley, "Solutions to klee's rectangle problems", Technical report, 1977.

[4] D. Beyer, "Competition on software verification", 2012, 504–524.

[5] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, "Symbolic model checking without BDDs", 1999, 193–207.

[6] M. K. Buckland, F. C. Gey, "The relationship between recall and precision", *JASIS*, **45**:1 (1994), 12–19.

[7] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, "Counterexample-guided abstraction refinement", *CAV*, 2000, 154–169.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph", *ACM TOPLAS*, **13**:4 (1991), 451–490.

[9] A. Groce, R. Joshi, "Extending model checking with dynamic analysis", *Verification, model checking, and abstract interpretation*, 2008, 142–156.

[10] ISO, The ANSI C standard (C99), ISO/IEC, 1999.

[11] F. Ivančić, S. Sankaranarayanan, "NECLA static analysis benchmarks", 2009.

[12] D. Kroening, A. Groce, E. Clarke, "Counterexample guided abstraction refinement via program execution", *Formal methods and software engineering*, 2004, 224–238.

[13] C. Lattner, V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation", 2004, 75–86.

[14] D. Novillo, "Tree SSA: A new optimization infrastructure for GCC", *Proceedings of the 2003 gCC developers' summit*, **21**, 2014, 83–93.

**DOI:** 10.18255/1818-1015-2015-6-763-772

# Эффективное исполнение программного кода в контролируемом окружении как способ улучшения результатов статического анализа и верификации программ

Беляев М.А., Ицыксон В.М.

*получена 15 сентября 2015*

Существующие средства и методы статического анализа и верификации кода на языке C используют различные методики упрощения программной модели, приводящие к значительному снижению точности анализа. В данной работе представлен новый подход к повышению точности анализа путем исполнения программной модели в контролируемом окружении, которое позволяет точно определять ошибочные ситуации, такие, как нарушения контрактов кода и ошибки работы с памятью, оставаясь при этом эффективным с точки зрения затрат по времени и по памяти. Данный подход был реализован в модуле под названием «Tassadar» в рамках средства ограниченной проверки моделей «Borealis». Прототип был опробован на стандартных наборах тестовых программ данного средства и показал минимальное влияние на его общую производительность.

Статья представляет собой расширенную версию доклада на VI Международном семинаре “Program Semantics, Specification and Verification: Theory and Applications”, Казань, 2015.

Статья публикуется в авторской редакции.

**Ключевые слова:** программная интерпретация, символьное исполнение, статический анализ программного кода, точность анализа

**Для цитирования:** Беляев М.А., Ицыксон В.М., "Эффективное исполнение программного кода в контролируемом окружении как способ улучшения результатов статического анализа и верификации программ", *Моделирование и анализ информационных систем*, **22**:6 (2015), 763–772.

**Об авторах:**
Беляев Михаил Анатольевич, orcid.org/0000-0003-1260-9211, ассистент
Санкт-Петербургский политехнический университет им. Петра Великого,
194021, Россия, г. Санкт-Петербург, Политехническая ул., 21
e-mail: belyaev@kspt.icc.spbstu.ru

Ицыксон Владимир Михайлович, orcid.org/0000-0003-0276-4517, доцент
Санкт-Петербургский политехнический университет им. Петра Великого,
194021, Россия, г. Санкт-Петербург, Политехническая ул., 21
e-mail: vlad@icc.spbstu.ru