

Equid—A Static Analysis Framework for Industrial Applications

Maxim Menshikov(✉)

Saint Petersburg State University, 7–9, Universitetskaya nab.,
St. Petersburg 199034, Russian Federation
info@menshikov.org

Abstract. The rise of the software engineering industry sparkled the research on static analyzers in both academia and industry. Academic tools historically have an exhaustive feature set but don't easily apply to industrial applications, and industrial verifiers are still very limited. The Equid project, which loosely stands for "Engine for performing queries on unified intermediate representations of program and domain models" is an attempt to fill the gap between theory and practice by building a language-agnostic analyzer in close contact with development and security community. In this introductory paper we set project goals, reveal motivation and describe code processing stages, such as preprocessing, translation to project's own intermediate codes, virtual machine execution, constraint solving, all done to make static and interactive contract violation checks easier, more precisive yet informative. The project is compared to other analyzers. We believe that such a framework can draw attention to industrial uses clearly missed by verification communities and help shape a vision of universal static analyzer architectures.

Keywords: Static analysis framework · Virtual machine · Intermediate representation · Type system · Multi Solver · Generalized Syntax Tree · Language-agnostic analysis

1 Introduction

Developing large software packages comprising numerous tools, especially when depending on specialized hardware, is hard. A number of bugs can occur at different abstraction layers—operating system core, shell, network setup, userland or even hardware levels. Searching for an error in such an environment is tedious, and static analyzer's abilities in this field are limited by the authors' vision. In many situations, built-in universal detectors are not enough to detect and fix the issue. Moreover, extinguishing the error fire is pointless if the analysis is not integrated into the development environment. Inability to find a satisfying utility becomes a major reason behind stopping using verification at all [9]. Academic tools aren't easy to use and thus are hard to enforce at a corporate level. With rich software "forward"- and reverse engineering experience, we aim to fill this

© Springer Nature Switzerland AG 2019
S. Misra et al. (Eds.): ICCSA 2019, LNCS 11619, pp. 677–692, 2019.
https://doi.org/10.1007/978-3-030-24289-3_50

gap by making our own framework designed for industrial applications. This paper is the first attempt to present our perception of static analysis field.

Project's goal. Make a static analyzer which would fit industrial applications, be universal and customizable to a reasonable extent. Support C and RuC [24] languages.

Paper's goal. Introduce an analyzer framework and its internals to a wider audience, describe the approaches to memory modeling and code execution. We understand that it is impossible to cover all aspects of the tool within a single paper well enough, nevertheless, having a starting point for consequent papers is highly desirable.

Motivation. Authors are not aware of the tools doing precise yet refinable analysis and integrating properly to the development of industrial applications: the tools are usually black boxes. The RuC (a safer alternative to C) support suggested by A.N. Terekhov required a tunable verifier to suit current and future language features.

Novelty. The project is an example of unique architecture for performing various kinds of analyses based on virtual machine framework. The focus on memory operation precision through the type system. The Multi Solver, combining SMT and Abstract Interpretation results. RuC support.

The paper is organized as follows. An architecture for multi-language support and flow- and context-sensitive analyses is outlined in Sects. 3 and 4, the solver and type system design is explained in Sect. 5. The common infrastructure is shown in Sect. 6. The whole solution is evaluated in Sect. 7.

1.1 Industrial Applications

In the quest for supporting industrial applications, the development is based upon the following observations:

- Industrial tools are usually large and debugged to some extent, the occurrence of memory corruption and other undesired events is unlikely. That makes the extensive state space exploration redundant. Of course, some critical components might need it.
- Static analysis tools in the best case must be *supplementary to the programming process*: it is hard to prove *all* the algorithms automatically, so the developer should help the tool. As large code bases tend to suffer from ownership problems (nobody besides the author can understand what is going on inside, sometimes even he can not), the contracts are a natural extension to the documentation.
- Contracts—not only functional but also memory access, etc—are usually enough to find the majority of observable bugs.

– There are cases when state space exploration on demand is required. The trivial example is using buggy third-party modules, which are for some reason mandatory (e.g. platform drivers). We won't cover queries in this paper as this is definitely worth an additional investigation.
– The static analyzer has to support "night analysis", load distribution and must be easily invokable.

This leads to a conclusion that a good tool is not necessarily able to prove all the properties, but should rather help the developer write a readable, better documented, more reviewable code. However, it doesn't mean the support of state-of-art techniques is omitted, we are all hands for it and develop them, but it might be impractical to focus on it before getting more feedback from current users.

2 Operation Schema and Classes of Detectable Errors

The project is schematically depicted in Fig. 1 (explanation follows). It is aimed at finding a few specific error classes:

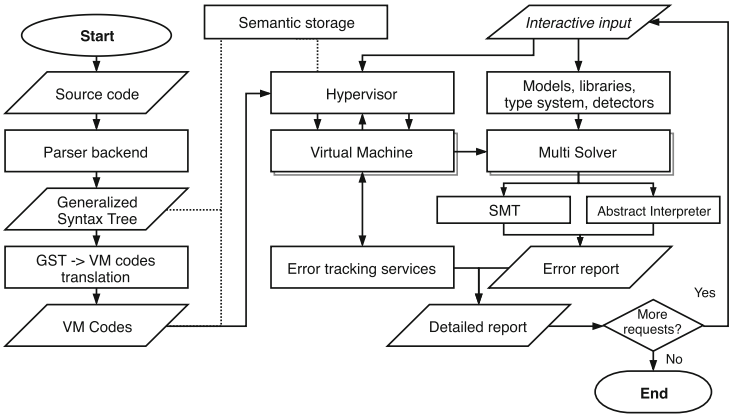


Fig. 1. Full analyzer's operation flowchart

1. **Contract violations**. This is done on the basis of the model checker generating SMT formulas out of the virtual machine operational codes.
   Advantages of the analyzer in this area: the arbitrary level of memory abstraction precision (through the use of CPU/Compiler/type abstractions).
2. **Memory safety errors**. This class of errors is detected by a combination of the model checker and abstract interpreter.
   Advantages: controllable memory abstraction level. Precision is modifiable.
3. **Always reachable/unreachable paths**. Such error conditions are detected by using a model checker.
4. **AST errors, e.g. duplicate operands**. This is detected on the Generalized Syntax Tree (GST) level. Advantages: GST detectors may get insight from other levels.

The project uses a notion of a Semantic Storage. This entity is responsible for saving, retrieving and grouping of Resources, Fragments, Expressions, Virtual Machine codes and other artifacts of the analysis process, saved in an internal format and augmented with key-value metadata. It is essentially a MongoDB database running the background. The use of the database is not mandatory as long as the artifacts may fit into the RAM, large projects may get a substantial stability improvement using it. The aim is to cover it extensively in a separate paper, but parts of it are mentioned in [19].

The other benefit of the analyzer is that it brings lingual agnosticism: analysis is the same for any language and adding a new one is not time-consuming. All kinds of analysis are glued up through common communication means. Objects have properties, such as name, invocation context, GST/SMT mapping, etc, which are saved to a semantic data storage and are accessible at an arbitrary time by developed detectors, which in general work on the top of Virtual Machine operational codes.

3 Processing Architecture

3.1 AST Support

The project doesn't bind to any specific framework such as LLVM [17]. Instead, Clang [2] is only one of possible *parser* backends. The module creates an Abstract Syntax Tree by Clang means, and *transformers* convert it to *Generalized Syntax Tree*. The GST comprises many expression types forming a program *basis*—it includes typical constructs like branches, binary/unary expressions, etc. A transition to GST removes syntax sugar, which is also helpful in removing complexities of development and intercommunication.

Integrated development environments prefer bijective transformers for refactoring since modified objects have to be represented in AST form again. Although static analyzers don't have such restriction—the GST suggests the use of surjective transformations, saving of mapping within nodes is nonetheless helpful for printing purposes. Consider A a set of all possible Abstract Syntax Tree nodes, G a set of all possible Generalized Syntax Tree nodes. Then $f : A \to G$ is a transformer from AST to GST.

From a technical standpoint, the A is heavily bound to parser's implementation. The GST and transformation schema impose a few implicit restrictions on input parsers. They are enforced within Roslyn and are feasible for Clang.

1. All AST nodes must be inherited from the same base class or be distinguishable anyhow. As Clang declaration and statement nodes are distinct, an additional wrapper around A is formed.
2. Each AST node must have a parent and explicit children.

3.2 Languages

The analyzer supports two languages: C and RuC. LLVM/Clang provides support for C, and the transformation between LLVM/Clang and GST is currently

mostly 1-1. The second language, besides obvious Unicode keywords (which are easily removable), supports syntax sugar for threads. The initial support is added via decomposition to functions invokable in place.

The design of the GST generally supports C#, so the applicability to object-oriented programming languages is trivial, although such use requires a mangling of method names and injection of metadata to GST. Functional programming languages require GST improvements. However, due to the industrial use background, the project focuses its efforts on C-like languages.

Another important role of GST is to mine useful persistent relations between program statements. Consider $X = (x_1, x_2, ..., x_n)$ an ordered set of same-level statements, where $x_i \in X$ $(i = 1...n)$ is a single statement. If there exists some property separating a set to two distinct (possibly, empty) ordered sets, it is possible to find a separator $F(X) = \{(x_1, x_2, .., x_{n_1}), (x_{n_1+1}, ..., x_n)\}$, where x_{n_1} is a separator statement. At the end of this operation, we get two ordered sets X_1, X_2. If there are multiple properties, the rule is to separate sets sequentially.

A one-pass analysis might employ a conservative sequential algorithm. Consider $X_1, X_2, ..., X_m$ ordered sets produced on previous stages (in a degenerate case, X_1 might be the only set, and it might be empty). Then each new statement forms a single-element set X', which is either appended to X_m in case there is no separator for (X_m, X'), or renamed to a set X_{m+1} used in next iterations.

These sets are called *fragments* in the analyzer. Fragments are a convenient way to group statements by various signs, e.g. the convention is that any structural statements (branches, loops) reside in isolated fragments by default, each *locking* call such as **spin_lock** immediately changes the intercommunication environment. Lockset-like analyses are exceptionally trivial because of that. Also, fragments may have designated **goto** labels, a single parent and a number of children. The next fragment in execution order, as well as sibling, can be calculated in a runtime. The GST augmented with such information is called *extended*, and it provides a number of advantages to the traditional AST approach: the Lockset-like analysis, the simplification of processing.

The expressions in Fragments are additionally simplified by means of the analyzer. While this may be considered bad practice since every real-world compiler optimizes and reorders instruction set itself, relying on the other entity like LLVM is even more dangerous because it still suggests one specific untraceable behavior. The manual simplification of Fragments gives a possibility to at least add meta information that expression is potentially dangerous.

4 Virtual Machine Framework for Flow and Context Sensitivity

The Virtual Machine framework is *by the very nature of it* a symbolic execution engine yet in a traditional, not program analysis sense. It keeps a track of entry point sets, executes Virtual Machine codes and maintains the resource aggregation. All the *actual* analysis is outsourced to the Solver framework, meaning that Virtual Machines do no more than just a provision of flow- and context-sensitivity. In this section, we define high-level means used to make it so.

4.1 Virtual Machine Language

Extended GST is not a convenient form for a flow-sensitive analysis, although it can be used to find simple mistakes. Understanding a real execution order in presence of syntax sugar (e.g. **goto**, **break**) requires a different representation.

LLVM IR and similar intermediate codes resembling machine execution is the usual solution for this issue. During the early evaluation the following problems were observed for an environment without intermediate code between LLVM → SMT/other verification means:

1. Loading/storing various analyzer-specific globals is impossible without making an IR augmented with some other kind of instructions, or introducing phantom variables.
2. Using third-party project limits parsing to what it supports. Interpreted languages have a good chance of being easier convertible to less hardware-like IR.
3. It doesn't give a chance to introduce custom types not mimicking hardware types.

Project's IR is free of these problems and is bound to flat GST expressions (their grammar is not enforced by IR). IR reveals expression's key properties, i.e. what does the expression actually mean to the analyzer. Part of possible operation code kinds are listed in Table 1 (actually, more kinds are supported). The IR is also bound to semantic storage behind: every referenced resource has ID [19], making it possible to save or unload them dynamically, reducing the memory footprint. This allows using a distributed technology stack and simplifies the process for all consequent analyses.

4.2 Hypervisor and Virtual Machines

A hypervisor is on the top of hierarchy [19], it manages virtual machines running the programs in VM language. When a flow-sensitive analysis is requested, the Hypervisor inserts function summaries (generated using Craig interpolation [18]) or annotations instead of *invoke* calls. In context-sensitive case, a sequence of *enter*/*exit* operations with function body unrolled to some extent is constituted instead of *invoke*. While simplified extended GST mostly deals with constant values and references, the result of a function call is saved to a designated $Variable$ set by *enter* command. By enumerating functions during the execution and flexibility, the Virtual Machine-based framework is fairly effective in achieving flow and context sensitivity.

5 Solver Architecture

The solver's architecture is a core of the whole analyzer. The key to effective operation lies in combining the abstract type system with the results of multiple fairly different solvers.

Table 1. Virtual Machine language operational codes

Command	Description	Example
declare	Fetch a resource from semantic storage and add it to the current visibility area	*declare resource*
assign	Assign a value to a variable	*assign* $x = y$
branch end branch	Add a condition for next operational codes before enclosing *end branch*	*branch* $x > 5$ *assign* $z = 5$ *end branch*
check	A condition which is determined to be always false. The body is never processed	*assign* $x = 0$ *check* $x > 0$
constraint	A condition which is determined to be always true. The body is always processed	*assign* $x = 0$ *constraint* $x = 0$
invoke	Invoke a function with a given set of parameters and save the result to a variable	*invoke* $res = f$ *with arguments* (x, y)
system	Invoke an internal command. Similar to *invoke*, except that internal functions are called by name, not by resource handle	*system assert with arguments* $(x > 0)$
enter exit	Start a new visibility block, setting where to save results. This function is heavily used for inlining. Must be finished with *exit*	*enter store result* *to* z

5.1 Type System

Literals and variables have straight-forward types. Unary expressions mostly resemble the type of their subexpression: parentheses, increments. **Dereference** and **address-of** expressions change inner type indirection level. Ternary operations in C require second and third operands to be of the same type (but it is not a strict requirement in other languages). The hardest are binary expressions which have a very detailed description in C standard [1].

The GST contains a type field in each node whenever it is applicable. In other cases, a static analyzer may deduce the type from expression's semantic meaning (defined by the target language) and subexpression types.

The other problem is that machine representation for unambiguous types can be different (mostly varying by endianness in the real world), and compiler chooses the most adequate representation for ambiguous types, e.g. plain **int** might be signed or unsigned, 2^n bit long. The CPU choice might limit the integer to 32 bit or any other target architecture register size. Any analyzer which tends to be type-precise, not necessarily supporting type reinterpretations, has to take this into account. For Equid, a variety of low-level types was created, as

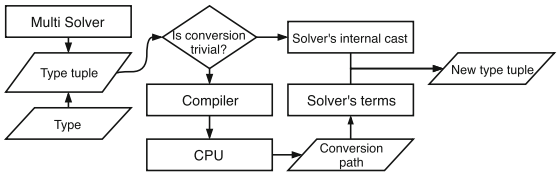


Fig. 2. Type conversions in the presence of cheap and expensive methods

well as their symbolic and numerical conversions to and from the *widest* types, all glued together via CPU and Compiler classes for various vendors/versions. Precise conversions are costly, so each solver might employ a different cheap schema for trivial cases, e.g. CPU endianness means not much for properly coded applications. Those which don't obey the compatibility rules might enable costly abstraction and decrease the efficiency of the whole analysis process.

The schema of type conversions is provided in Fig. 2. Mentioned *tuples* are introduced in consequent paragraphs.

5.2 SMT Solver Backend

Theorem solver supporting major SMT features is essential to an analyzer: it is used for contract violation checks. The project turns the whole function(s) to a set of *assert* clauses and checks the negation of the *goal*.

The reason for choosing CVC4 [6] as the main solver is that it is robust and provides a well-defined native C++ interface. The latter is notable since intensive input/output operations may have an impact on performance. The number of theories supported by CVC4 is impressive [4]. The infrastructure is not limited to this solver, however, major efforts are put into separating CVC4 responsibilities from the rest of the analyzer. They include:

- Effective assignment *access path* generation must be done in target solver's framework. Access path in terms of our project is a variable with a collection of transformations, required to actually assign a value to that variable. To understand why is it important, consider the following example. $array[x] = y$ in terms of imperative languages is only suitable for initialization, the real assignment is slightly different: $array = old_array\ WITH\ [x] := y$. In more complex left expressions, e.g. $array[x].member1[y][z].member2$, the need for access path is more obvious.
- The generic type system doesn't depend on the solver, but cheap casts must reuse as many solver's abilities as possible—they include truncation/widening of bit vectors, built-in casts operation. Also, as there is no direct mapping between C types and CVC4 types, so they are passed inside the analyzer in tuples (t_s, t), where $t_s \in T_s$ is a solver's type and $t \in T$ is a real type. This way the cast operation can always perform the conversion from (T_s, T) to (T_s, T'), choosing the most appropriate T'_s.
- Predicate system is implemented entirely in solver's terms mainly because not all solvers actually need predicates.

5.3 Abstract Interpreter

Built-in abstract interpreter augments the SMT formula of the main solver with the knowledge retrieved in the form of abstract domains—interval or polyhedral (still in development). At one hand, it interprets the Virtual Machine codes, produces an over-approximation of the semantics. This over-approximation is passed to the Model Checker, causing it to reduce the state space for SMT [11], which results in improved precision. On the other hand, a secondary solver like this is capable of detecting errors on its own, especially when combined with a described type system and symbolic/numeric domains developed for these purposes. A trivial abstract interpreter is also used for pointer analysis. The abstract interpreter is actively participating in CEGAR [10]-like analysis. In our tests, it doesn't have any significant influence on analysis time.

5.4 The Multi Solver

The power of both solvers is combined in one Multi Solver (MS). The main idea is that SMT solver and Abstract Interpreter collect their own data sets and save it to MS context, which is extremely volatile because of its flow/context-sensitive nature. The processing is as follows (Fig. 3):

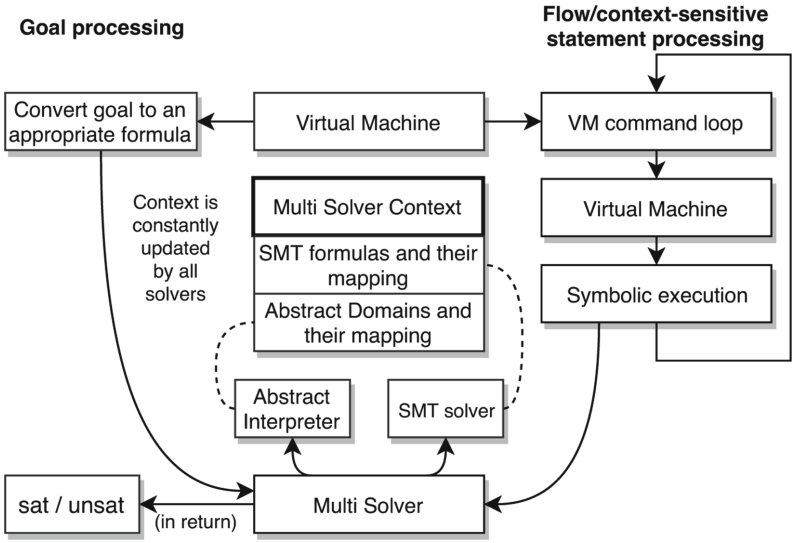


Fig. 3. Multi Solver operation scheme

1. Each virtual machine creates a context based on rules provided by Hypervisor.
2. VM codes are processed by VM itself.
3. VM codes are passed to Multi Solver:

   (a) *Branches, loops*: MS processes such commands by wrapping the inner commands into blocks, excluding them if they are unreachable. At the end of branch/loop processing, MS wraps inner commands to the conditions more correct from **else if** point of view.
   (b) Other commands are passed to SMT/AI solvers. The SMT solver produces SMT formulas for each program statement and AI solver generates abstract domains. This information is saved to the SMT/AI context both of which are parts of MS context.
4. Goals are passed to Multi Solver:
   (a) The MS chooses the most convenient interpretation of formulas for the specific case. There are cases when unaltered abstraction is superior (e.g. when analyzed program state space is reasonably small), in other cases, like when the loop is potentially boundless or at least not proven to be different, the overapproximation replaces it.
   (b) The sat/unsat result is provided.

Our implementation may also refine the abstraction, but this part is not ready for presentation. We found the described algorithm effective in real-world applications.

6 Common Infrastructure

6.1 Detectors

Contracts for user-provided and standard library functions are among the possible ways to find real issues. The other feasible way is writing a detector. The detector is largely an LTL formula defining flow-sensitive properties. Although in the project formulas are encoded in a common fashion, a few detector types can be outlined:

1. GST visitors. Such detectors make it possible to find duplicate operands, etc.
2. Resource/variable markup manipulators. **malloc** allocates data marked up as *allocated*, **fopen**'s result is either null or an opened handle. A trivial detector for memory or handle leak would simply check that any object which is not passed outside the function is freed at the end of a function.
3. Sequence observers. User-provided script might present a template for a bad sequence, e.g. **fclose(x)** followed by **read(x)**.

The actual way to write out such scripts will be demonstrated in consequent papers as the language is finalized.

6.2 Standard Library

A standard library for C and RuC is physically located in separate header files, which are preloaded to a semantic context on each analyzer's start. We have a prototype recompiling annotations to C code with over sixfold faster load, thanks to missing parser invocation. The processing of library functions is a

MapReduce-like approach [21], in a sense that they are by default tagged as *non-existent* and are replaced with real implementation if the user overrides it.

Annotations are provided in ACSL form[1]. The project features an ANTLR [23]-based parser transforming the input to GST, therefore all solver/virtual machine APIs can take advantage of the ACSL just like the annotations are simply statements inside the code. The assert clauses are directly mapped to *assert* VM commands and preconditions/postconditions turn to check/constraint/nop commands based on the context: obvious omissions are made for start entry points, a currently verified function, etc. The contracts overall provide an extremely fast initial bug search.

7 Evaluation

In this section, we check the resulting framework synthetically and practically.

7.1 Syntax Construct Support

The syntax support requires teamwork of many modules, and the following syntax constructions (beyond simple ones) work in the project:

1. The GST natively supports real functions and generally promotes their native invocation. Self-modifying code is strongly prohibited, and call indirection works only within visibility range, which might be set in globals, within a single function or within investigated call chain. Thus, the analyzer is not applicable for the cases with many accessors. There is a workaround for the problem—a declarative pointer analysis, in which pointer targets are set within ACSL comments.
2. Branches and loops are processed within the bounded model checking framework, and when this is not enough to prove properties—abstract interpreter provides a safe abstraction of them. Still, it is not precise and we aim at the unbounded processing.
3. Array support involves a kind of shape analysis with array summaries. This ensures the precision in the most common cases (e.g. sequential array reads) but is extremely overapproximating in random access tests and large objects. The abstraction might turn to a more precise one in case the object is reasonably small (the actual size is configurable).
4. Unions are not supported properly. The support is generally provided (on the basis of memory abstraction), but no major efforts had been put into making it work properly.
5. The **goto** instruction is very widespread in C programs for safer resource deallocation in error cases. It is supported by Fragments and transparently propagated to all layers. Jumps (e.g. **setjmp**) are supported, but their use is discouraged because they break the fragment model.

[1] https://github.com/maximmenshikov/acsl-grammar.

7.2 Common Errors and Contract Violations

Significant efforts were put into testing software on real many-purpose industrial projects. Our example included a 500 KLOC operating system management application.

For the project, domain-specific rules were added in the form of ACSL function models. The following modifications were required:

1. It was determined that in such specific use cases it is required to add a *declarative* pointer analysis. The core of the application is considered stable, however, a number of ways to perform indirect function call complicate analysis. Declarative pointer analysis is based on the notion of function aliases, e.g. */feature/x/enable* tree-like name instead of cryptic internal names. After such a modification analyzer was able to find common contract violations.
2. The tested project also had weaknesses undetected by any other analyzer (e.g. uninitialized structure members) due to cryptic module structure. With the verifier aware of it, all weaknesses were detected. While the efforts for supporting such project were significant (a few days to support the whole project), most of the modifications were needed due to the immaturity of the analyzer.

Of course, numerous errors were found in the analyzer during the evaluation. The project suffered from all sorts of null pointer dereferences and other unpredicted situations, which is not uncommon [7]. For example, the project was not expecting nested functions to appear that often and therefore lacked an entry point selection for that case.

7.3 Toyota ITC Benchmarks

In the author's master thesis, the analyzer was tested against a supported subset of Toyota ITC benchmarks. The actual versions at the moment of writing were **Clang 3.9**, **Frama-C Silicon**, **cppcheck 1.76**—for this test, we reused the results from [20]. The *w_Defects* part was analyzed and compared to other analyzers. A solid percentage of detections was achieved—like 90% of bugs in supported categories (Table 2). For all such cases, the project acts similarly to Frama-C, sometimes worse, sometimes a bit better.

However, the test doesn't include inter-procedural analysis and many other aspects, so the evaluation is incomplete. A bigger evaluation is scheduled after planned functionality improvements.

Table 2. Error detection count with Toyota ITC benchmarks

Test	Equid	Frama-C	Clang	cppcheck	Total
bit_shift	17	17	14	11	17
buffer_overrun_dynamic	30	32	1	2	32
buffer_underrun_dynamic	35	39	2	3	39
data_lost	19	3	—	—	19
data_overflow	25	16	—	9	25
data_underflow	12	8	—	5	12
littlemem_st	11	11	—	—	11
null_pointer	15	16	13	12	17
overrun_st	47	54	2	21	54
ptr_subtraction	2	1	—	—	2
underrun_st	13	13	2	5	13
uninit_pointer	10	16	11	5	16
zero_division	16	16	13	8	16

7.4 Querying Code

One of the project's ongoing development processes is deep querying of software properties. The querying language is not yet ready for public presentation, therefore it is a subject for disclosing in consequent papers. By and large, the language is an LTL which augments Virtual Machine codes, which are later proven by Multi Solver.

The author managed to apply it to a limited scope of a large 3rd party Linux kernel driver (500+ KLOC) and find reasons for errors in it. It required some efforts to build proper models—but no automatic search tools were able to catch most of the problems in such a badly structured code. It can be considered a success.

7.5 The Computational Cost of the Approach

At this early stage, we were unable to properly verify how the approach compares with other analyzers in terms of computational cost, yet we *estimated* if the costs are at least sane. Yet in our preliminary estimations, the time to analyze files is comparable with Frama-C [16].

Looking through the process, there are a few basic steps. *Source to GST transformation* doesn't add up significantly because it is almost stateless (*Resources* are the part of GST, but there are relatively little of them compared to expressions). *GST to VM code* mapping is not cheap, although it is hard to find an analyzer which doesn't do this in any form (GCC/LLVM [17] IR, CIL [22]). So, this can be considered as not harmful. The algorithm doesn't overestimate SMT and tries to reuse as many computations as possible. Thus the cost of the whole solution solely depends on the abstraction, which can be refined.

The RAM/ROM usage may go higher when using distributed technology stack [19]. This is expected and therefore not harmful.

8 Related Work

There is no deficiency in this research area.

The CPAChecker [8] platform is by far the closest project ideologically. It aims to make the analysis configurable just as our project does, additionally, it provides interfaces to SMT solvers, interpolation, octagon and BDD libraries. CPAChecker only supports C. It uses CIL [22] language as an intermediate representation. Frama-C [16] is great regarding precision and informativeness [20], and it is full of plugins. Also, our project uses the same format for annotations—ACSL. SPIN [14] is specifically bound to checking models in the specific format. We prefer a smaller and more readable format for industrial applications which won't result in writing a completely separate model, but it is impossible to underestimate the contribution of this model checker to a verification field.

Svace [15] is an effort of the Institute of System Programming to make a universal static analyzer for C/C++/Java/C#. A lightweight analysis is started for each language, the main part generates a call graph and then a set of checkers test routines against known problems. We find the analyzer polished, with incremental analysis and a web interface for results. The project is battle-tested in Android and Tizen operating systems. Borealis [5] is one of the examples with similar technical means—it features a Model Checking combined with Abstract Interpretation, just like Equid. However, it is based upon LLVM IR, which is closer to metal than our language. Absence of GST limits input languages yet simplifies the initial development. The goal of our project is a bit shifted towards industrial applications.

The intermediate representations have gone a long way. The most famous IR is LLVM [17], which is essentially an assembler-like language. SAIL [12] is providing a good IR-to-source mapping and is used for almost the same purposes as our language—reducing the need to reason about complex expressions. The good point about it is an integration with GCC which results in low overhead. REIL [13] is more focused on disassembling input files. Frama-C [16] and CPA Checker [8] use modified CIL (C intermediate language). Even judging by its name, it is limited to C language. Roslyn [3] (.NET Compiler Platform) shares ideas of GST (in a sense that common syntax nodes can be used for both C# and Visual Basic.NET), and the Common Intermediate Language is a distant analog for Virtual Machine Language, although not made specifically for static analysis purposes. Overall, no project specifically aims at using separate semantic storage, additionally simplifying the use of distributed technology stack.

9 Future Work

Although the first results are promising, there is still a lot of work to do. The author is looking towards improving abstract interpreter. More advanced domains are in progress, a stricter abstraction refinement for type system is also being developed. The solver system requires a more complex pointer analysis than implemented at the moment of writing. Further augmenting the Virtual

Machine language with environment/resource model data is a way forward to the whole proving process.

Another part to care about is threading support. It is important as is for all languages, but is essential [25] for covering a complete set of RuC functionality. It was done in the first version—a race condition finder but was largely forgotten in the new version. In the end, the analyzer needs countless improvements on library annotations, the full project runs, etc.

10 Conclusion

The paper introduced Equid, a static analyzer for industrial applications. Main algorithms used in the analyzer are presented. The parsing approach is based on the notion of Generalized Syntax Tree, which is a unification of C/RuC abstract syntax trees. The Hypervisor/Virtual Machine framework for flow- and context sensitivity is presented. We describe our solution for verifying function contracts, a part of Multi Solver combining Abstract Interpretation and Model Checking to one cooperating framework. The analyzer was successfully tested in the telecommunications industry and compared to the known tools, with reasonable success in supported categories. As there are not many frameworks for industrial purposes, we believe that bringing a configurable solution benefits the community and might draw attention to the functionality required for the industry.

References

1. C11 Standard ISO/IEC 9899:2011. https://www.iso.org/standard/57853.html
2. Clang: a C language family frontend for LLVM. http://clang.llvm.org
3. Roslyn -.NET Compiler Platform. https://github.com/dotnet/roslyn
4. SMT-COMP 2018 Results. http://smtcomp.sourceforge.net/2018/results-summary.shtml
5. Akhin, M., Belyaev, M., Itsykson, V.: Borealis bounded model checker: the coming of age story. In: Mazzara, M., Meyer, B. (eds.) Present and Ulterior Software Engineering, pp. 119–137. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67425-4_8
6. Barrett, C., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
7. Bessey, A., et al.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (2010)
8. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
9. Christakis, M., Bird, C.: What developers want and need from program analysis: an empirical study. In: 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 332–343. IEEE (2016)

10. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_15
11. Cousot, P., Cousot, R.: Refining model checking by abstract interpretation. Autom. Soft. Eng. **6**(1), 69–95 (1999). https://doi.org/10.1023/A:1008649901864
12. Dillig, I., Dillig, T., Aiken, A.: SAIL: static analysis intermediate language with a two-level representation. Technical report. Stanford University (2009)
13. Dullien, T., Porst, S.: REIL : a platform-independent intermediate representation of disassembled code for static code analysis (2009)
14. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
15. Ivannikov, V., Belevantsev, A., Borodin, A., Ignatiev, V., Zhurikhin, D., Avetisyan, A.: Static analyzer Svace for finding defects in a source program code. Program Comput. Soft. **40**(5), 265–275 (2014). https://doi.org/10.1134/S0361768814050041
16. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. Form. Asp. Comp. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
17. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, p. 75c. IEEE Computer Society (2004)
18. McMillan, K.: Applications of craig interpolation to model checking. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 22–23. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30124-0_3
19. Menshchikov, M.: Scalable semantic virtual machine framework for language-agnostic static analysis. In: Distributed Computing and Grid-technologies in Science and Education, pp. 213–217 (2018)
20. Menshchikov, M., Lepikhin, T.: 5W+1H static analysis report quality measure. In: Itsykson, V., Scedrov, A., Zakharov, V. (eds.) TMPA 2017. CCIS, vol. 779, pp. 114–126. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-71734-0_10
21. Menshchikov, M.A., Lepikhin, T.A.: Applying MapReduce to static analysis. Control Proc. Stab. **4**(1), 433–444 (2017)
22. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_16
23. Parr, T.: The definitive ANTLR 4 reference. Pragmatic Bookshelf (2013)
24. Terekhov, A.N.: Programming and compiler techniques educational tool. Comput. Tools Educ. **1**, 36–47 (2016)
25. Terekhov, A.N., Golovan, A.A., Terekhov, M.A.: Parallel programs in RuC project. Comput. Tools Educ. **2**, 25–30 (2018)