TRUST IN SOFT

TRUSTINSOFT EBOOK - SEPTEMBER 2020

UPGRADE YOUR EXISTING TEST SUITE WITH TRUSTINSOFT

By Jakub Zwolakowski

Introduction

All serious developers and project managers pride themselves on the quality of code they create and maintain. Acutely aware of the disastrous consequences which may arise from unnoticed bugs and vulnerabilities (e. g. Heartbleed, Toyota Recall 2018), they put tremendous effort into ensuring that their software is robust, correct, and secure.

During long busy days, with caffeine rushing through their veins and the cool breeze of air conditioning blowing in their faces, they steadily navigate through the treacherous maze of pull requests in front of them. Composed and careful, they never advance into uncharted territory without thorough preparation. They move forward only if the way is deemed safe: after meticulously designing proper test cases and doing their very best to steer clear of any foreseeable hazards.

But their duty does not end there! During long sleepless nights, in a dim glare of computer screens and soft hum of cooling fans, these valiant men and women persist by their posts. Despite the fatigue they stare incessantly at the sea of code they feel responsible for, their sharp eyes scrupulously reviewing each line with focus and perseverance, scanning for any sign of danger...

Such is the fate of these stalwart individuals, ever compelled to carry the heavy burden of responsibility and uncertainty! This is especially true for those working with C or C++ (by far the most fearless and resolute of all developers), who are accustomed to wrestling an element so incredibly powerful and volatile yet so often deployed where stakes are highest, and peril abouns. The single chilling question repeated in the backs of their minds whenever their code is deployed to production: “What if I missed something?”.

Because they know that no matter how many test suites were prepared and executed, how strictly coding guidelines were enforced, to what extent sanitizers were employed, and how many hours were spent carefully reviewing the code, this risk remains very real: they might have missed something.

Yes, despite all their endless sweat and toil, there can never be a complete guarantee for safety and security of their software!...

Despair no more, brave developers and gallant project managers! TrustInSoft Analyzer provides such guarantees. Strong mathematical guarantees, based on formal methods and backed with decades of scientific research. Our solution relies on exact techniques (like abstract interpretation) in order to prove properties of programs written in C/C++. No more uncertainty, no more hoping for the best! Instead, hard evidence that allows us to actually finally trust the software.

You might have heard about formal verification. It is a wholly different paradigm than verification by testing. On the one side, the level of assurance it provides, concerning both safety and security of the analyzed code, is radically superior compared to other available methods. On the other side though, formal verification of a program is considered to be a difficult, expensive, and time-consuming process which requires skill and experience. Due to the high cost, in terms of both time and effort, this approach does not suit everyone: according to Wikipedia, formal verification may be even as much as 80 percent of the total design cost. This is why up until now it was mostly used only for particularly critical pieces of code.

TrustInSoft opens a brand-new door into the marvelous realm of formal verification, providing a way of entry accessible for every developer! We make this powerful technique considerably more attainable thanks to our hybrid solution, as we equip developers with tools that empower them to gradually lift a C/C++ project from the level of verification by testing to the level of formal verification. By virtue of these tools and the accompanying methodology such a feat can be achieved with minimal added effort, mostly by leveraging already existing test suites.

Essentially, we strive to provide a light and easy access to the many blessings of state-of-the-art formal verification techniques without the need to perform a heavy and difficult full-scale analysis of the code traditionally required to obtain such results.

Content

Systematic approaches to developing correct software

When one wishes to develop a correct program, writing a set of tests is usually the first solution which comes to mind. And it is a pretty good one! A set of rudimentary tests is a quick, simple, and effective means to rapidly find the most glaring faults in the code. However, a set of manually crafted test cases can only get us so far. The problem of building safe and secure software must be eventually tackled in a more systematic manner if the goal is to achieve a reliable outcome. There are several distinctive perspectives and approaches which allow advancement on the glorious path towards correct code.

Automated testing / test generation

The efficacy of testing itself can be vastly improved using dedicated methodologies and tools.

We may employ various techniques of automatic or semi-automatic testing or test generation to greatly enhance test coverage and pertinence. For example, fuzzing is a highly adaptable and efficient method for providing invalid, unexpected, or random inputs for a given program for testing purposes. Using tools that implement such techniques is an excellent idea which we recommend wholeheartedly.

The important thing to keep in mind though is that no matter how advanced a methodology or a tool is applied, the result, basically, is more tests and / or better tests. And unfortunately it is impossible to test each and every existing scenario of non-trivial program's execution one by one, there are too many of them.

Testing can be compared to probing the waters to check for rocks (or sea monsters), which may hide under the surface, before veering forward.

Dynamic checking tools

Now, one step further from testing there is dynamic checking. Tools that implement this approach are known as sanitizers. Verification still happens mostly on the level of executables: sanitizers help to determine whether the program is engaging in any dangerous behaviour when being executed. Their added value, on top of traditional testing, is that suchtools reveal some hidden problems which do not impact test results or directly cause runtime errors.Two main techniques are employed to track what a program is actually doing under the hood. The first technique is to insert additional instrumentation into the original source code during compilation (some sanitizers work simply as compiler extensions, e.g. for Clang we have ASan, UBSan, MSan, and TSan). The second technique is to simulate the program's execution in a virtual environment controlled and monitored by the sanitizer (e.g. Valgrind).

As both these methods allow for a deeper level of assurance about the correctness of a program's behavior on particular test cases, sanitizers can be roughly summarized as testing on steroids. We perform the same tests as usual, but we discover more problems. We do not only see what is happening on the surface of the water (i.e. check the test's results), we can also peek a little bit below the waves to get an idea of what lurks beneath (i.e. check if some dangerous operations were performed during the test execution). Probe the same points, detect ten times more krakens! Nice!

"This is Yellow Submarine reporting to the Command! We've got a visual on that deadly Giant Octopus. It is lurking in the shade, in what seems to be some kind of a garden, well hidden under the sea... Requesting immediate debug at our coordinates!"

Using sanitizers is thus a direct improvement over simple testing and as such it is highly recommended. Their main shortcoming is that even though such tools discover many faults, they are still not exhaustive, therefore can miss some serious problems. Sanitizers are, in essence, debuggers: they help to find and understand faults in the code, but they will never be able to guarantee to eliminate all the faults.

Pattern-matching tools ("traditional" static analysis)

Another interesting set of tools are the so-called linters. Linters work directly on the source code level and use pattern-matching, and sometimes other traditional static analysis techniques, in order to find and flag potential problems. Depending on their sophistication, these tools can detect many different classes of issues: obvious programming errors, violations of style guidelines, usage of suspicious constructs, etc. Using a linter is a great way to improve general code quality and spot bugs. Moreover, this category of tools is pretty handy when striving to reach conformance with a particular coding standard (e.g. Misra, Cert C) especially in the area of style-related rules and recommendations.

The caveat here is buried in the main assumption underlying these tools: the assumption that by making code nicer, we will also make it more correct. Of course, neither making code nicer (e.g. elegantly written, well structured, abundantly annotated) nor blindly following any coding standard actually guarantees correctness. That said, following style guidelines or coding rules and writing elegant code is actually not a half bad idea. Keeping everything tidy and manageable may not be the ultimate solution to all problems, but it definitely helps eliminate some immediate errors, and yields undeniable maintenance-related gains in the long run.

Every sailor knows that you should keep your boat clean and orderly. This is not for aesthetic reasons, but for safety! Order makes issues easier to spot and thus helps to avoid potential disasters. Of course, it does not stop a rope from snapping. Yet, it definitely improves a sailor's chances of noticing that a rope is fraying or that it has snatched onto something. And this increases the odds of preventing the problem in time or at least allows for the mitigation of consequences.

Correct from the ground up

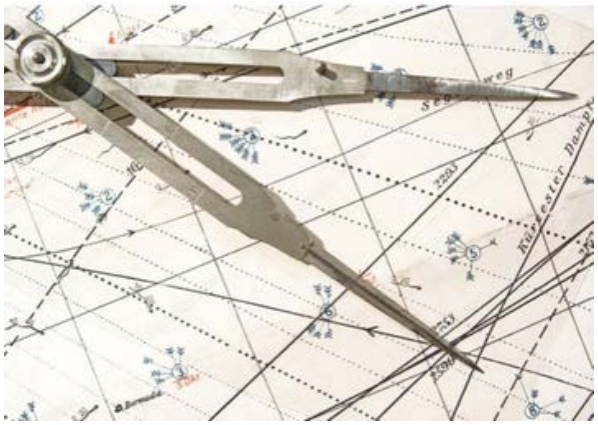
One radical approach to eradicating bugs and vulnerabilities is to not make them in the first place. There are several technologies and frameworks that aim at designing and building correct software from the ground up. For example, the SCADE Suite employs a formally-defined domain-specific language to express what the program should do and then it generates executable C/C++ code that is correct by design. Some other noteworthy examples are the B method, used for safety-critical systems, and Cryptol, used for cryptographic algorithms.

These are extremely effective approaches, which render it virtually impossible to introduce implementation-level faults in the program. And, in an ideal world following one them would be recommended whenever possible. Unfortunately, in practice such techniques come with substantial constraints which make them suited exclusively for very strict development processes like those for writing embedded critical software. Moreover, they are definitely not applicable to existing programs, at least not unless rewriting them completely is an option.

So, unless you are ready to ditch your magnificent tall ship and build a cable ferry instead, this is hardly the solution for you.

Full formal verification

Rounding up this list, there is the idea of fully verifying software formally. Formal verification of a piece of code is undeniably superior to both testing and pattern-matching, as it provides actual mathematical guarantees concerning the software's safety and security properties. And with a tool like TrustInSoft Analyzer such an undertaking is within the realm of feasibility (even if it remains a rather ambitious task, especially for a beginner).



Maybe, you know, use maths instead of YOLO?...

Formal verification is much like creating a mathematical model of the ship, analyzing its behavior with respect to the sea, and proving some properties about it (e.g. that it is sea-monster-proof). Which is in fact not so far from what we do when designing boats, cars, bridges, or other physical objects used by people!

Important aspects of such objects, like material endurance versus weight to bear, are mathematically determined and calculated using models based on our understanding of the laws of physics. Doing the same for programs seems nothing but sane and rational.

Hybrid solution

The solution we advocate proposes to boldly go in the direction of formal verification, but it strives to eliminate the daunting difficulty of performing such a verification to the full extent. To decrease this difficulty we will be aiming a bit lower (narrowing the verification's objectives) and piggybacking on existing groundwork where possible (which cuts down the amount of the extra work needed).

So, instead of attempting to formally verify a C/C++ program with all the bells and whistles, our goal is to benefit from as many advantages of formal verification as possible with minimal effort. We achieve it with TrustInSoft Analyzer by following a specific methodology based on leveraging existing test suites.

Even taking a single small step toward formal verification of a piece of code is precious. It unlocks access to results and benefits that are unachievable by simple testing or even dynamic checking. As in our approach formal verification is fueled by the tests, it can be combined perfectly well with all the test generation techniques which we mentioned before, further amplifying their efficiency. In addition, completing the first step opens the door for potentially continuing the route to complete formal verification

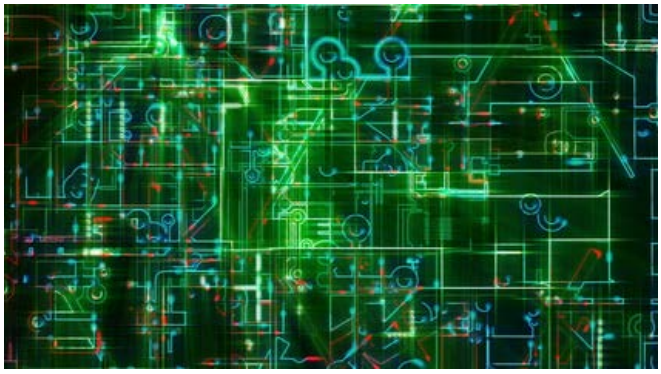
What is TrustInSoft Analyzer? What does it do?

The TrustInSoft Analyzer is a powerful and versatile tool used for advanced verification of C/C++ software. It can determine with mathematical certainty whether the provided code is safe and secure by detecting undefined behaviors or proving the absence of those behaviors.
Here, we use the Analyzer in a very specific way, though: we tailor it for a much narrower and more specific purpose. Thanks to this particular configuration, called the Interpreter Mode, we can circumvent all the complexity traditionally involved when performing a full-blown analysis, and we can apply the Analyzer almost directly to existing test suites with minimal setup effort.

Difference

When software is tested, it usually means that the compiled program is executed on a computer, and its actual behavior is compared to how it was expected to behave. In other words: the program is fed specific inputs (defined by given test case) and its outputs are checked for errors.

TrustInSoft Analyzer does not execute a compiled program natively on the machine where it runs. Instead, it works on the program's source code level, interpreting it line by line and simulating the program's behavior in a virtual environment. Based on complete formal understanding of the C/C++ language semantics and a complex model of the computer's memory, this simulation is mathematically sound.



When I write phrases like "complete formal understanding of the C/C++ language semantics and a complex model of the computer's memory" I imagine something like in this picture, blinking and buzzing, and I nod my head knowledgeably, feeling very smart for a short moment.

What does it mean? The program is no longer a black box. All the details and fine points of its internal behavior become observable. Now, not only the program's outputs can be checked for errors, but everything that happens during the program's execution can be thoroughly examined and verified for signs of trouble. In difference to the dynamic checkers, this verification is sound and exhaustive: we detect all the faults without fail. Yes, all of them.

Why is this important? Because programs are vicious little clever beasts. They are fully capable of committing most atrocious, forbidden, and dangerous things. And they can hide their crimes so well that they can go unnoticed for a very long time. Moreover, these monsters do not even feel guilty about it... And then one day, when you go down to the basement you stumble upon all these cadavers stuffed in the dark corner, and your program just goes "But why are you mad? I was not supposed to do that?". So, do not trust them. Install cameras in your proverbial basement. Be safe.

In other words (coming back from whatever happened in the previous paragraph!...), it may happen that the program's naughty behavior does not cause a runtime error and does not change the test's result. In this case it is not observable in any way during testing. It may even be that such behavior is masked by the compiler's optimization or it triggers only on a particular target architecture in specific circumstances. In this case it is not detectable by a sanitizer either. And just like that, because of unfavorable circumstances, the problem might get overlooked. A bug or vulnerability will remain concealed, lurking somewhere in the program, waiting to be exploited...

Writing outside of array's bounds is a perfect example of such a situation.
In some cases, out-of-bound writing will not cause a runtime error and will not alter the program's output.
Maybe it will just change the value of some random object in a way that very rarely impacts the program's behavior. Maybe the whole write operation will even get completely optimized away during compilation. And then, one beautiful day, some unfortunate event or a malevolent hacker discovers a combination of parameters and environment variables which causes this out-of-bound access to be executed in a way that actually does something consequential and... well, the day is not so beautiful anymore!

Guarantees

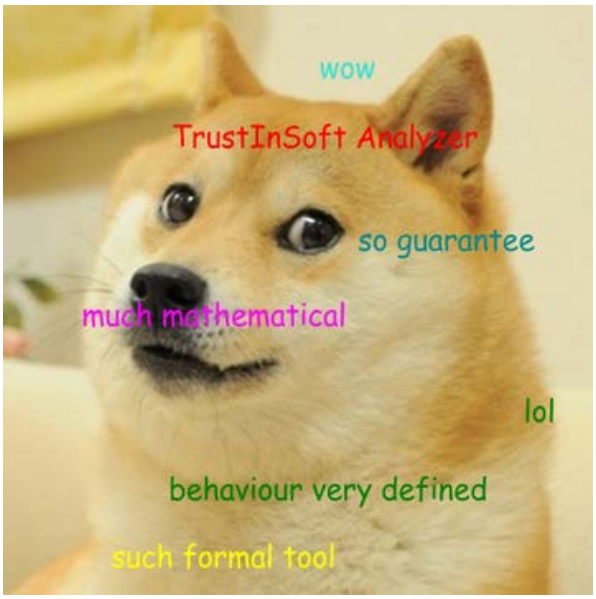
TrustInSoft Analyzer detects if such dangerous behaviors can happen during the program's execution. In fact, it is capable of doing much more than just detecting them. When the Analyzer concludes that certain faults do not appear in a program, this does not only mean that it cannot find any.

This means that it has mathematically guaranteed the total absence of a certain category of problems in the perimeter of given test cases.

And this is not an uncertain and meek “well, umm, so I’ve searched around a bit and I could not really find any more problems, so I guess we should be safe now” situation. This is a serious and reassuring “There are no problems left, Sir! We have taken care of them all, Sir! You can trust me on that, Sir!” situation.

Deployment

Now, enough talking about the wonderful advantages of TrustInSoft Analyzer (which is easy to use, formally sound, fully exhaustive, soft to the touch, and provides incredibly strong mathematical guarantees). Let us discuss what is necessary to deploy this magnificent tool on an actual C/C++ project.



Should Doge become our Brand Ambassador?

Requirements

Source code: complete C/C++ projects

TrustInSoft Analyzer works directly on the source code level. In order to carry out a meaningful analysis of a program, it requires access to all the C/C++ source code that the given project uses or includes. The tool does not make guesses about the source code, it only works within a well-defined context.

This constraint restricts the Analyzer applications to complete C/C++ projects: the code of all the dependencies must be available. Otherwise, an adequate stub for every called external function whose source code is not available must be provided. In such a case the setup unfortunately requires some additional effort. But that is the price of exhaustiveness.

Existing test suite

The methodology presented here is based on leveraging existing test suites. We analyze the program's behavior by performing abstract interpretation of all the available test cases. Thus a test suite with significant coverage greatly increases the profitability of this approach.

Know-how: to compile and build the project

When working on a complex multi-file project, TrustInSoft Analyzer needs to know how the whole project is compiled and built: which source code files are used, which headers should be included, what compilation options should be set, etc. This is necessary in order to properly parse and analyze any complex code.This know-how might have different shapes and forms.

Sometimes all the instructions about building and compiling are just written in plain English in a single README file. Usually though some kind of a build system is used (for example the project comes with a regular Makefile). Either way, when setting up TrustInSoft Analyzer, all the relevant pieces of information must be extracted from these sources. Luckily, some helper tools that facilitate this process (e.g. by intercepting all such parameters during the program's standard build procedure) are available.

Run, correct, rerun, guarantee!

Now, the Analyzer can be run on each of all the tests which constitute the program's test suite. If undefined behavior is detected in a test, the Analyzer provides a specific warning. The underlying program faults are investigated and corrected. Then, the Analyzer is run again and again to discover subsequent problems, each of which is corrected one by one, until no more undefined behaviors are found in the code.And when the tests finally pass through the Analyzer without any warnings, this means something quite amazing. It means that the corresponding execution paths in the source code are guaranteed to be 100% free of undefined behaviors. Mathematically proven, pinky promise!

So what now? What happens when the whole test suite passes through the Analyzer without raising any alarms?

Well, that's it! The job is done! Now you can finally sit in that soft, comfortable armchair in front of the glowing fireplace, open that priceless bottle of old bourbon, light that exquisite Cuban cigar, and rest! And try to enjoy some well-deserved peace of mind for a while... Or immediately get back to work, because you hate comfy furniture, open flames, alcohol, smoking, idleness, and generally being relaxed! Arrr!

So, three major axes for improvement present themselves at this point:

- **First: incorporating TrustInSoft Analyzer in your continuous integration efforts**. All the difficult groundwork has just been prepared, so if you already have some continuous integration activities going on, mixing in the Analyzer should be straightforward. Currently we are in the process of developing a dedicated continuous integration service for projects hosted at GitHub, stay tuned!
- **Second: extending the existing test suite**. Writing new tests, or generating them using a dedicated tool, now will not only augment the test coverage, but also the analysis coverage. All the execution paths added to the analysis perimeter will be guaranteed clear of undefined behaviors.

- **Third, moving towards more complete formal analysis**. With TrustInSoft Analyzer, the existing test drivers can be generalized until the analysis perimeter stretches throughout the whole program, and exhaustive verification is reached.

The infamous three major axes for improvement. Ba-dum-tsss...

Summary

All serious developers and project managers pride themselves on the quality of code they create and maintain. Acutely aware of the disastrous consequences which may arise from unnoticed bugs and vulnerabilities, they put tremendous effort into ensuring that their software is robust, correct, and secure.

Testing is not enough to verify software that matters for security and safety. Formal verification can satisfy such high concerns, but requires a significant effort for deployment. It is rarely used on non-critical software. Other techniques (automatic test generation, dynamic checking, pattern-matching static analysis, etc.) are pretty good ways to complement or improve testing, but do not address the core of the problem.

We propose a new hybrid approach, based on deploying TrustInSoft Analyzer on existing C/C++ projects, and leveraging their current test suites for immediate gains. This provides the benefit from the strong mathematical guarantees associated with formal methods, in the whole perimeter covered by the test suite, without investing all the time and effort needed to fully verify a program formally.

Hopefully, such a solution will help make formal verification more accessible and allow more C/C++ developers to enjoy its ample advantages. They will start sleeping better at night and living happier lives, relaxed and serene, knowing that their programs are finally free of bugs and vulnerabilities. And what is more gratifying than safe and secure software? A smile of bliss on a developer's face, of course!