An Evaluation of General-Purpose Static Analysis Tools on C/C++ Test Code

Jean Malm, Eduard Enoiu, Masud Abu Naser, Björn Lisper
School of Innovation, Design and Engineering
Mälardalen University
Västerås, Sweden
firstname.lastname@mdu.se
masud.abunaser@mdu.se

Zoltán Porkoláb
Faculty of Informatics
Eötvös Loránd University
Budapest, Hungary
gsd@caesar.elte.hu

Sigrid Eldh
Ericsson AB
Stockholm, Sweden
sigrid.eldh@ericsson.com

***Abstract*—In recent years, maintaining test code quality has gained more attention due to increased automation and the growing focus on issues caused during this process.**

Test code may become long and complex, but maintaining its quality is mostly a manual process, that may not scale in big software projects. Moreover, bugs in test code may give a false impression about the correctness or performance of the production code. Static program analysis (SPA) tools are being used to maintain the quality of software projects nowadays. However, these tools are either not used to analyse test code, or any analysis results on the test code are suppressed. This is especially true since SPA tools are not tailored to generate precise warnings on test code.

This paper investigates the use of SPA on test code by employing three state-of-the-art *general-purpose* static analysers on a curated set of projects used in the industry and a random sample of relatively popular and large open-source C/C++ projects. We have found a number of built-in code checking modules that can detect quality issues in the test code. However, these checkers need some tailoring to obtain relevant results. We observed design choices in test frameworks that raise noisy warnings in analysers and propose a set of augmentations to the checkers or the analysis framework to obtain precise warnings from static analysers.

***Index Terms*—testing, static analysis, test maintenance, fault detection, code quality**

I. Introduction

Automated testing is essential in software quality assurance projects [1]. This requires trust in the test's quality. Despite this reliance, test quality is not enforced to the same degree as on production code [2] [3]. Test artefacts are generally given less attention than production code during the code review process. Contributing factors include the lack of time given by management and unfamiliarity with the test code in question [4]. This creates technical debt through issues manifesting as e.g., *flaky* tests. Tests are naturally checked for functional correctness by running them enough times to gain confidence of their stability. In comparison, extra-functional properties such as clarity or adherence to coding standards are largely left to manual code review.

Static program analysis (SPA) tools can automate the review and refactoring process. But based on issues identified by Spadini et al. [4], this is not done for test code, despite available tooling [5].

Tools guide the review effort by flagging potential issues in the code. Developers may thus focus on more difficult-to-automate issue fixing. Tools additionally fill the role of an expert system, as code- or certification standards may be encoded in their ruleset[1]. For common cases, tools can even suggest fixes, either for developer's approval [6] or as automated patches [7].

Such reviews are only as good as the tools themselves. Some scepticism regarding the usefulness of SPA tools lives on [8]. Common issues raised include noisy reports (false positives) and a lack of suggestions/guidance for fixing issues [8] [9]. While automation narrows down the work for developers, time is still spent on report review [10] and configuration of analyser options, as the default configuration may not be fully sufficient [11].

Given the wealth of existing tools targeting different types of code quality aspects, one must state that despite design differences, *test code is also code* [12]. For tool selection in industrial environments, the trade-off between cost and value immediately becomes a deciding factor [13]. Familiarity and proven reliability of already-used tools are likely less costly to adapt and extend. Additionally, open-source tooling allow practitioners to extend them with company- or domain-specific checks, as such they represent extensive engineering effort [14]. Therefore, it is well-worth investigating how static analysis tools used for production code work for test code.

To evaluate the usefulness of general-purpose analysers on tests we analysed a sample of 10 open-source C/C++ projects using three analysers fitting the criteria of being open-source and used in practice: Clang-Tidy, Clang Static Analyzer, and CppCheck. We evaluated the warnings generated by these tools to identify useful checks that the tools possess already and discuss amendments for making them more suited for usage on test code. We thereby answer the following research questions:

RQ1 What are the quality issues of *test code* identified by general-purpose static analysers?

This work is supported by Vinnova, Sweden's innovation agency through the ITEA3 Projects TESTOMAT, XIVT and SmartDelta. Eduard Enoiu has been supported by the Horizon 2020 program (No. 957212).

[1]For example, Carnegie Mellon University maintains a list of analysers with SEI CERT C++ coverage at https://wiki.sei.cmu.edu/confluence/display/cplusplus/CC.+Analyzers

RQ2 How serious are the identified issues for tests?
RQ3 What modifications to the static analysers and test code are necessary to utilise the potential of SA on test code?

We identify several design choices of tests and test frameworks that cause noisy warnings from these tools, some marked as high-severity. Notably, many warnings of null pointer dereferencing were raised by the analysers' inability to recognise custom test assertions.

A vast majority of warnings relate to code style violation. They become noisy as test frameworks commonly use preprocessor macros and the warning may be triggered by something in the macro expansion.

The rest of the paper is organised as follows. Section II provides the background discussing the details of the static program analysis and the properties of the test code. Section III provides the related work, IV and V gives the details of the tools and study setup to facilitate the replication of our study. It also discusses our data set, data extraction and evaluation. Section VI includes the description and explanation of our results. The results and validity threats are discussed in Section VII. Section VIII concludes the paper along with presenting further work.

II. Background

In this section, we provide an overview of static program analysis and present some properties of test code that may benefit or hamper its analysis.

A. Static Code Analysers

Static analysis techniques extract properties from a given software without executing it. The technique has many uses, from compilers to heavy-duty program verification[2]. They are usually a complement to the dynamic approaches in continuous integration, such as executing a set of tests. A benefit of static approaches is that it is applicable on parts of the software that may not be executed in isolation. Given a fast enough analysis, expert knowledge may be encoded to help developers enforce coding guidelines in real-time through automated code review [15].

Unlike tests that run on concrete software and hardware, static analysis uses abstraction formats: abstract syntax trees (AST), control-flow-graphs (CFG), or program dependence graphs. This allows the analysis to scale and may give some assurances w.r.t. absence of issues, but instead introduces the potential of raising false positive warnings.

Pattern-based analyses look for predefined patterns in the code. This type is suitable for finding syntactical or structural issues, such as style-based violations or missing initialisation. Pattern-based analyses are comparatively fast, as they rely on only processing the statement type(s) related to the issue.

Semantic-based analyses require accounting for the underlying execution behaviour. Some modelling of state during execution is required. Such analyses target issues requiring specific execution sequences, such as usage of undefined variables, potential divisions by zero or memory leaks. Such analyses often track data and execution flow through the program. Popular approaches to perform such analysis include *abstract interpretation* where a set of data-flow equations are set up and then solved using some fixed point iteration [16]. On the other hand, techniques such as *symbolic execution* will traverse some exploded version of the CFG (e.g., generate separate nodes for each iteration and branch) and keep track of symbolic values. During the node generation and traversal, it also generates path constraints which are then fed to a constraint solver in order to detect which paths are feasible to traverse. This way, it may emulate execution while still being able to reason about a program on a symbolic level [17].

To be useful in non-trivial software, this tracking shall handle function call semantics, so-called inter-procedural analysis. In some cases function calls may cross into different files (a.k.a. translation units). If the function is not familiar to the analysis, it errs on the side of caution yielding some too-broad result. One solution is to perform a cross-translation-unit (CTU) analysis. The analyser pre-processes available function definitions to extract and store details for later use. Existing approaches utilise both procedure summaries [18] [19] and inlining of functions. While this improves the accuracy of results, such pre-processing increases both memory usage as well as execution time [20].

B. Test Code Design - Impact on General-Purpose Checkers

Software tests are a set of instructions where the goal is to put the system in some given state, act upon that state, and then verify that results match expectations [21]. Tests are grouped together into *test suites* based on logical or practical reasons. Both these properties are common enough that they are universally supported by *testing frameworks*.

This design impacts the usage of general-purpose static analysers targeting test code. Dedicated suite-wide setup functions are used to prepare test objects before each test, such as (re-)setting a database to some common state. This promotes code reuse, but introduces an implicit execution order. General-purpose tool may lack this knowledge ordering and assume execution orderings that lead to irrational warnings.

The separation of test and production code is also true from a project layout perspective. Based on documentation of test frameworks, standard practice is to separate tests from the translation unit (e.g., code file) being tested [22] [23]. This reduces the analyser's ability to reason about the code across file boundaries. It reduces precision by treating unfamiliar function calls as a black box. Analysers will then typically err on the side of caution and fall back on e.g., unconstrained results. Thus, we assume CTU analysis to be required to get good results when analysing test code.

Automated software tests utilise *assertions* to verify expectations against actual behaviour. These check some condition and aborts the execution if the condition is false. As the test would otherwise terminate, it may be assumed that immediately after the assertion, whatever is asserted must hold. Some

[2]For a comprehensive tool list, the authors recommend https://github.com/analysis-tools-dev/static-analysis

analysers use this fact to increase the accuracy of the analysis. By design assertions occur naturally in test code. Hence such code *should* be well-suited for analysers that utilise them. In fact this idea has been applied even to reduce false positives by inserting and model checking assertions against warnings [24].

III. Related Work

Many tools providing dedicated static analysis for finding test code issues exist. A recent mapping study presented a number of test smell detectors created until 2020 [5]. The authors determined that from the set investigated some form of static analysis was the most used approach for detection. They additionally noted the large number of tools supporting Java, the duplication of smells detected and the lack of tools featuring refactoring guidance. Due to the specification of smells, issues are generally detected by extracting metrics [21] [25] or pattern-matching on the code [26] [27]. In addition to smell detection, static analysis is also applied as part of flakiness detection tools [28]. While our work focuses more on the gap between general-purpose analysers and test code, the analysers presented may be used as inspiration.

The Java static analyser FindBugs was evaluated on tests [12]. The authors did this by extracting a set of Java tests reverted to a known buggy state and then check if the tools found the bug. The result noted the tool did not detect any of the issues, despite having some checks that are closely related. While we are interested more in causes and fixes of perceived issues, replicating this work would give stronger evidence for the usage of general-purpose tools in this context.

To help practitioners deal with reports, automated program *repair* (and refactoring) approaches have been documented both from academia [29] [30] and in *grey literature* through blogs [31].

IV. Tools

Based on discussions with industrial collaborators, we chose three C/C++ static analysers from open source that are already in use by their organisations[3]: Clang-Tidy, Clang Static Analyzer, and Cppcheck. Specifically, we utilised Clang release 15.0.0 for Clang-Tidy and Clang Static Analyzer and Cppcheck 2.7 for our data collection. Of course, this is not an exhaustive list, but these tools represent commonly used techniques used in code review and bug finding.

A. Clang-Tidy

Clang-Tidy [32] is the LLVM compiler framework *linter*. Linters are static code checking tools that warn about issues such as poor style choices or usage of bug-prone behaviour. It follows from *extreme programming* practices that code review feedback is most useful while the code is being developed [33]. Otherwise, developers may move on to other tasks, motivations/design choices forgotten. Linters are a way to provide such feedback automatically. The tool works on the Abstract Syntax Tree (AST). Thus it is most suited for structural- and style-related issue detection. Automatic fixing of issues is supported through suggestions of code changes, which we remind the reader is a request from practitioners for for adoption in practice [9].

Clang-Tidy consists of a set of *checkers* divided into groups depending on their intended use, such as *performance* or *bugprone*. They are designed as pattern matchers on the AST, where matchers are constructed using a declarative-style domain-specific-language (DSL)[4]. Each matcher is bound to some callback function triggered when a match is found, in order to take further actions.

The DSL provides a simplified way to interact with the clang AST. It may be argued this makes it possible for stakeholders of different levels to contribute in specifying automatic checks and fixes. This automation alleviates maintenance and refactoring efforts of industrial-scale code-bases [31].

B. Clang Static Analyzer

The Clang project provides a development framework for creating checks requiring path-sensitive functionality: Clang Static Analyzer (Clang SA) [34]. To reduce the classical scaling issues of symbolic execution [17], Clang SA applies some heuristics and path pruning (e.g., loops by default are traversed a maximum of four times). Clang SA checkers collaborate by producing and consuming nodes in the exploded graph. This allows for integration and modelling of e.g., critical library-specific functionality by simply adding a new checker.

Assertions can be used to drive the symbolic exploration. Clang SA has automated detection mechanisms for custom asserts but it is not fool-proof [35]. Test assertions are similar in behaviour to C/C++ standard asserts. However, they are implemented only to terminate a single test case. Depending on the approach, this may not be detected by the analyser. Instead, optional checkers exists that model detection and extraction of conditions from test assertions. At the time of writing, this is only done for GoogleTest assertions in official releases of Clang SA.

C. Cppcheck

Cppcheck is a program analysis tool that focuses mostly on finding undefined behaviour [36]. For simple rules, Cppcheck provides configuration files for writing checks using regular expression matching. Alternatively, checkers use a combination of unsound dataflow analysis and AST-traversal to check the resulting code for issues.

Cppcheck ships with a set of knowledge configuration files. These files contain specifications of well-known libraries, with information about the behaviour of functions and their arguments. These configurations include knowledge of a number of test frameworks, among them GoogleTest and Boost.

V. Study Setup and Design

We gathered some random projects for study using Github[5]. The selection criteria was $>$ 1000 stars AND $>$ 100 forks

[3]Representatives from telecommunication and automotive domains

[4]For an up-to-date reference, see e.g., https://clang.llvm.org/docs/LibASTMatchersReference.html

[5]https://github.com/search/advanced

TABLE I
PROJECTS USED FOR EVALUATION. FOR METRICS, WE GIVE A BRIEF DESCRIPTION, WHICH LANGUAGE(S) ARE REPRESENTED AND THE LINES OF TEST CODE (EXCLUDING HEADER FILES). PROJECTS FROM THE CURATED SET IS MARKED BY A STAR.

Name	Description	Language	Test code (kLoC)
Thrift⋆	Cross-language services	C/C++	54.2
Xerces⋆	XML Parser	C++	16.0
Bullet3	3D physics engine	C/C++	43.9
CMake	Build System	C/C++	37.7
LLVM⋆	Compiler framework	C/C++	136.8
Firefox⋆	Web browser	C/C++	511.7
OpenCV	Computer vision	C++	96.3
Subsurface	Diving logger	C++	47.0
Surge	Sound synthesizer	C++	5.4
VTK	Image processing	C/C++	402.7

TABLE II
USAGE OF C/C++ STATIC ANALYSIS PER PROJECT.

Name	Used SA?
Thrift	Clang-Tidy (Also Coverity)
Xerces	No
Bullet3	Clang-Tidy
CMake	Clang-Tidy, Cppcheck
LLVM	Clang-Tidy, Cppcheck
Firefox	Clang (with Mozilla extension), Cppcheck (some)
OpenCV	Clang-tidy, Cppcheck (Singular reports)
Subsurface	Cppcheck
Surge	No
VTK	Clang-Tidy, Cppcheck

AND > 100.0 MB repository size, as well as C/C++ being the major languages and "test" being mentioned somewhere in the *Readme file*. Stars and forks were used as a crude metric for how wide-spread the project is, although it has be noted to increase naturally with an increase of collaborators [37].

Due to limitations on the resources for data gathering[6] and the manual nature of the review process, after some filtering to ensure presence of tests and build system, we ended up picking a sample at random to build. Building the projects was done to handle projects generating tests as part of the build process. Those would be missed if we relied only on e.g.., exported build system information.

In addition to the random sample, we extended the set with a curated selection used in previous work. In total, we present analysed test code from 10 different projects. The set of projects included in the paper can be seen in Table II, with the curated selection being marked with a star. In addition, we include information regarding the usage of C/C++ static analysers in the projects, based on configurations. Clang-Tidy has configuration files called ".clang-tidy", we may assume that if the file exists it has been used. For Clang Static Analyser and Cppcheck, we rely on the git commit log or bug tracker.

[6]Largely limited to one machine running Ubuntu 20.04, with 16GB RAM and an Intel i7-7820HQ 2.90GHz CPU

A. Analysis and Review of Test Code

CodeChecker version 6.20 was used for data gathering, as is the recommended way to perform CTU analysis in Clang [38]. It supports build-command capturing as well as managing and reviewing analysis reports from different analysers. Each tool was run in two configurations: *default* and full suite with CTU. Analysis reports were then parsed and stored in a CodeChecker server, each project-mode combination as a separate project. This allowed comparison of different modes and similar statistics to be extracted during the review.

The review process was done manually. The primary goal was to get a broad overview and find common causes of noise and issues with the findings of the tools. For each project we reviewed a random selection of each checker's warnings, with some care to spread the review over different files.

Commonly observed false or intended warnings examples were saved and potential solutions for filtering the noise were discussed internally. It shall be noted that the review was done within the author team. A more thorough approach would require reaching out to developers for further confirmation regarding the viability of reported issues.

VI. RESULTS

The number of hits separated by checker can be seen in Tables III, IV and V for Clang-Tidy, Clang Static Analyzer and Cppcheck respectively. For space reasons we omit the Low and Style severity from the tables, instead referring the reader to the repository. We note however those reports account for approximately 94% of all warnings.

A. Quality Issues Reported in Test Code

To answer RQ1, we summarise the warning reports based on reported severity levels.

Clang Static Analyzer reports a number of high-severity *core* checker violations. These model and produce warnings related to C/C++ core features. They should therefore be relatively stable, otherwise other results would be affected. Many warnings in the set relate to bad usage of memory and pointers. Furthermore, there are reports of array access issues.

Clang-Tidy high-severity warnings relate to error-prone usage of data types, such as implicit casts between integers and floating-point data. It also reports certain data access patterns, such as violating C++ move semantics.

For the medium-severity results, the tools reported a number of style-, precision- and performance issues. *Clang diagnostics* are reports from the compiler itself, mirroring gcc warnings. The majority of such warnings relate to unused parameters and uninitialised data in class objects, as well as potential misuse of data. In the majority of cases, this is caused by test framework- and utility functions, not test cases themselves. Clang SA additionally detects some issues with malloc and Unix-filestream (e.g., files not being closed).

The low-severity issues reported can be summarised to target some performance issues. These were cases where some sub-optimal function choices was made, or when unnecessary operations were performed. Cppcheck produced many style

TABLE III
CLANG-TIDY WARNINGS. TO REDUCE THE LONG NAMES GIVEN TO TIDY CHECKERS, WE USE THE FOLLOWING LEGEND. BP = BUGPRONE, CD = CLANG DIAGNOSTIC (COMPILER WARNINGS), CPPCORE = CPPCORE GUIDELINES.

Checker	Severity	All reports
BP-copy-constructor-init	Medium	9
BP-implicit-widening-of-multiplication	Medium	666
BP-incorrect-roundings	High	16
BP-infinite-loop	Medium	8
BP-integer-division	Medium	82
BP-macro-parentheses	Medium	748
BP-misplaced-widening-cast	High	36
BP-move-forwarding-reference	Medium	1
BP-multiple-statement-macro	Medium	5
BP-narrowing-conversions	Medium	2087
BP-not-null-terminated-result	Medium	25
BP-parent-virtual-call	Medium	5
BP-signed-char-misuse	Medium	46
BP-sizeof-expression	High	20
BP-string-constructor	High	1
BP-string-literal-with-embedded-nul	Medium	5
BP-suspicious-enum-usage	High	18
BP-suspicious-memory-comparison	Medium	10
BP-suspicious-missing-comma	High	3
BP-suspicious-string-compare	Medium	14
BP-too-small-loop-variable	Medium	2
BP-unused-raii	High	1
BP-unused-return-value	Medium	3
BP-use-after-move	High	112
cert-env33-c (std::system call)	Medium	2
cert-err33-c (unused return value)	Medium	1167
cert-flp30-c (float in loop)	High	43
cert-msc51-cpp (predictable random)	Medium	13
cert-oop54-cpp	Medium	103
CD-bitwise-instead-of-logical	Medium	1
CD-gnu-zero-variadic-macro-arguments	Medium	3
CD-implicitly-unsigned-literal	Medium	4
CD-main	Medium	1
CD-misleading-indentation	Medium	1
CD-missing-field-initializers	Medium	49
CD-unused-parameter	Medium	2060
CD-void-pointer-to-enum-cast	Medium	20
CPP-c-copy-assignment-signature	Medium	16
CPP-init-variables	Medium	5334
google-build-explicit-make-pair	Medium	21
google-build-namespaces	Medium	26
google-explicit-constructor	Medium	566
misc-definitions-in-headers	Medium	1025
misc-new-delete-overloads	Medium	1
misc-redundant-expression	Medium	8
misc-throw-by-value-catch-by-reference	High	24 (gen-cpp)
performance-move-const-arg	Medium	74
performance-noexcept-move-constructor	Medium	73

violations, mostly related to unused variables. In addition, style guides such as google and SEI CERT are included in the analysers triggered warnings.

> *Answer to RQ1: Based on our selected dataset and tools, general-purpose static analysis tools give a wide range of warnings on test code. However, the vast majority are style- and performance-related and are not classified as urgent by the tools themselves.*

TABLE IV
CLANG STATIC ANALYZER WARNINGS

Checker	Severity	All reports
alpha.nondeterminism.PointerSorting	Medium	7
alpha.security.ArrayBound	High	29
alpha.security.ArrayBoundV2	High	8
alpha.security.MallocOverflow	High	12
alpha.security.MmapWriteExec	Medium	1
alpha.security.ReturnPtrRange	High	1
alpha.security.cert.env.InvalidPtr	Medium	1
alpha.unix.SimpleStream	Medium	16
alpha.unix.Stream	Medium	38
alpha.unix.cstring.OutOfBounds	High	4
core.CallAndMessage	High	63
core.DivideZero	High	2
core.NonNullParamChecker	High	31
core.NullDereference	High	1824
core.StackAddressEscape	High	3
core.UndefinedBinaryOperatorResult	High	19
core.uninitialized.UndefReturn	High	1
cplusplus.Move	High	59
cplusplus.NewDelete	High	15
cplusplus.NewDeleteLeaks	High	17
cplusplus.PureVirtualCall	High	1
cplusplus.StringChecker	High	2
optin.cplusplus.UninitializedObject	Medium	6
optin.cplusplus.VirtualCall	Medium	32
optin.portability.UnixAPI	Medium	2
unix.Malloc	Medium	59
valist.Uninitialized	Medium	1

B. Severity of Issues

The tools gave a wide range of warnings on test code. We discovered that quite a few warnings were false positives or simply noisy. We address here the ones of higher severity or occurrence.

1) Null Access: The majority of the core checker issues reported are actually infeasible in reality. The main root cause is the analyser not recognising assertions properly. Recall that assertions will commonly terminate the test case with an error message if an assertion is false.

Clang recognises custom assertions based on functions being marked with the `noreturn` attribute. This is not typically done in test assertions. Problems occur when test assertions check that some pointer is not null. The result is that the analyser will try exploring both conditions, one of which the assumption is made that the pointer is null immediately after the assertion. Mozilla additionally uses null pointer dereference to crash the program intentionally. As we ran the checks on release builds, this was triggered essentially on every assertion. Based on the documentation, the intent behind the crash is to trigger a bug report from end-users to give developers more information.

2) Bugprone Checks: Clang-Tidy gave a range of "bugprone" hits related to potential precision- or performance issues. For instance incorrect rounding may result in incorrect behaviour. However, after reviewing the warnings, many of them turned out to be false positives.

The majority of use-after-move violations are done intentionally in order to test move semantics. Such tests will check

TABLE V
CPPCHECK WARNINGS

Checker	Severity	All reports
accessForwarded	Medium	1
accessMoved	Medium	5
allocaCalled	Unsp	27
assertWithSideEffect	Medium	20
assignBoolToPointer	High	1
constParameter	Unsp	90
constStatement	Medium	1
containerOutOfBounds	High	1
cppcheckError	Unsp	1
danglingTempReference	High	3
danglingTemporaryLifetime	High	1
duplInheritedMember	Medium	19
funcArgOrderDifferent	Medium	1
identicalInnerCondition	Medium	2
ignoredReturnValue	Medium	1
integerOverflow	High	8
invalidFunctionArg	High	2
invalidPrintfArgType_sint	Medium	114
invalidPrintfArgType_uint	Medium	74
literalWithCharPtrCompare	Medium	6
localMutex	Medium	4
memleak	High	4
mismatchingContainerExpression	Medium	5
mismatchingContainers	High	1
missingReturn	Unsp	37
negativeContainerIndex	High	1
noCopyConstructor	Medium	5
noOperatorEq	Medium	5
nullPointer	High	15
nullPointerArithmetic	High	16
nullPointerRedundantCheck	Medium	59
operatorEqVarError	Medium	21
qrandCalled	Unsp	3
qsrandCalled	Unsp	1
resourceLeak	High	1
returnDanglingLifetime	High	2
returnReference	High	1
returnStdMoveLocal	Unsp	3
returnTempReference	High	6
selfAssignment	Medium	1
shiftTooManyBits	High	1
shiftTooManyBitsSigned	High	15
sizeofFunctionCall	Medium	1
staticStringCompare	Medium	16
syntaxError	Unsp	26
uninitDerivedMemberVar	Unsp	876
uninitMemberVar	Medium	207
uninitMemberVarPrivate	Medium	12
uninitStructMember	High	4
uninitvar	High	5
unknownMacro	Unsp	17
unreachableCode	Medium	1
uselessAssignmentPtrArg	Medium	6

some state of an object before and after a move. The second check will naturally result in a warning, as accessing a value that has been moved results in undefined behaviour.

3) Miscellaneous Issues: In general, cases of unused parameter/function warnings are especially common in relation to mocking. This typically meant that only a few methods were actually implemented in practice. The rest would simply be stubs, either doing nothing or returning some base value, such as nullptr. In that case naturally the parameters are unused. The common fix for this is to simply comment out the names of the parameters in the argument list.

Clang SA found cases where files may be left opened. The issue was that tests were reading and asserting the content piece-by-piece. If an assert fails due to unexpected contents, the test execution stops, keeping the file pointer open. By first reading all of it into some temporary buffer, the file can be closed before performing the assertions.

We also note that Clang-Tidy comes with a check that looks for constant seeded random generators. In normal software that is considered a potential security risk as malicious users may use it for exploits. However, in testing deterministic behaviour is to prefer as it might otherwise cause flakiness if the randomness is dependent on e.g., time or random memory contents.

> *Answer to RQ2: Our results suggest that general-purpose static analysis tools can be used to detect several high-severity issues. Nevertheless, most of these issues are false positives or happen due to intended behaviour. To summarise, the lack of test-specific context information is causing issues by creating false-positive reports.*

C. Potential Fixes

Most warnings reviewed during the work were determined to be either noise or intended. Style- or Low-level severity issues are clear candidates for suppression. However, *core checkers* in Clang should not be turned off, and through suppression, we run the risk of missing real issues. Instead, we may rely on some test-specific assumptions to perform automatic clustering or refutation of reports, as has been suggested for usage of mitigating the manual effort [10].

The main culprit in Clang SA is the inability to identify custom assertions. This detection is detailed in documentation [35]. Extending the analysis core to allow users to provide information, similar to how Cppcheck works, would be one way. The analysis may then directly take advantage of the information. The downside is that such additions would require re-verification of the entire analyser in some contexts, and malformed configurations could lead to unintended consequences.

Clang and similar tools give justifications for produced warnings in the form of bug paths. Hence we may also post-process results to detect these cases:

- Warnings raised inside an assertion, as they are likely to be intended behaviour (e.g., testing error handling)
- Reports that continue on after a halting assertion has been failed. This is a mismatch between analyser and test behaviour.

Bug path nodes containing halting assertions may be identified based on e.g., user-provided configurations. Having that allows detecting cases where error handling is being tested: In the ideal case, the warning is raised in an assertion condition. This reduces to checking if an assertion part of the final node on the bug path. Warnings are by construction the final *control* node of the bug path, but may be followed by notes. If not,

we may backtrack to check whether it is part of a "death test" that is used to test termination [39]

Detecting warnings where Clang has explored an infeasible path is more straightforward given the context and data provided by bug paths. If it has continued on a faulty path, there is a halting assertion where Clang has assumed the assertion to be false. If a warning is raised at a later execution step that is not part of the assertion's evaluation, it is an indication that the warning is based on faulty information.

Test fixtures (primarily setup/teardown) cause noise w.r.t. uninitialised data. Recall that such cases occur as general-purpose analysers are not aware of the inherent call order between these test functions. A simple way to improve accuracy for certain checks would be to connect the test method with its optional setup and teardown. This may be done inlining the setup and teardown function bodies into the test cases. The benefit is that setup and teardown methods are typically well-defined based on naming conventions within a framework.

> *Answer to RQ3: Based on the root causes of noisy reports related to test code design we suggest a number of ideas for how to fix these issues. By post-processing the generated bug paths we may remove certain blatantly erroneous issues, reducing the manual time spent on reviewing such issues.*

VII. Discussion

Despite the numerous checkers offered by the tools, they yielded somewhat lackluster results when applied specifically to test code. There is some overlapping functionality between Cppcheck and e.g., Clang-Tidy, related to move semantics. Despite this, Cppcheck did better w.r.t. not reporting false positives. Cppcheck comes packed with predefined knowledge rules for at least Googletest and CppUnit, which are common frameworks encountered during the study. This makes Cppcheck suitable for usage on test code out-of-the-box.

The tools are fairly straightforward to extend with custom checkers, especially Cppcheck as this requires simply adding a configuration file. This allows augmenting them with checks targeting test issues, and reducing the false positives identified within this dataset. This would, in turn, allow for easier integration into existing workflows, compared to the effort of adding (and approving) entirely new tools.

Furthermore, as refactoring test code is an arduous task, the ability to suggest or even apply fixes automatically would allow us to reduce the effort spent on this task. In fact, several of the checkers triggered on tests came with concrete suggestions for how to improve the code.

A. Threats to Validity

A threat to the generalisation of results is the filtering process of projects under study and the set of tools used. We based the initial set on a few "popularity" metrics and then due to filtering and issues during the build process, the final set was reduced to 10 projects. While the results may not generalise to all domains, the projects selected include some well-known and mature applications.

There are threats to the reliability of the result obtained: The configuration or bugs in the underlying tools used may cause issues to be missed. In cases where this was determined to be user error, we updated the configuration for analysis and re-analysed the projects. Even for a reduced number of projects, the number of warnings generated too many reports to fully review manually. Instead of going through every warning, observations were discussed and documented at a high level within the author team. Potential issues may thus have been missed. To mitigate this, details regarding examples and mitigation approaches are provided online [40].

VIII. Conclusion

We investigated the usage of three state-of-the-art static analysers: Clang-Tidy, Clang Static Analyzer, and Cppcheck. The goal was to find out how well they behaved when applied to test code. In total, we ran the tools on ten different open-source projects. Based on the results we conclude that on test code the warnings were mainly noisy reports. To some extent this is natural due to lack of contextual knowledge on the tool's part; a lack of test behaviour modelling. Some issues derive from testing goals: The nature of tests is to expose potential issues in the code. Hence their implementation may violate established design rules to put the system in the necessary state. To augment the functionality of analysers on test code, we suggest some ideas to improve the developer's control of e.g. the sensitivity of checkers.

A. Future Work

We plan to expand the exploration with more tools and projects. Based on some encountered issues during the experimentation related to invalid configurations we propose investigating of how SPA analysis configurations could be inferred automatically (or at least warn practitioners of compatibility issues). For the tools evaluated, support for test code is still limited mainly to googletest framework. Extending it with some framework-agnostic modelling of testing semantics would be a follow-up. The tools studied offer support to implement a number of test smell detection identified in the work by Aljedaani et al. [5].

Clang specifically already provides a framework for automated refactoring suggesting, which would benefit users in large-scale systems. Cppcheck has numerous checks that could have fixes applied automatically, but to our knowledge, this feature is not available in the tool itself. Furthermore, the tool uses some annotations as part of its knowledge database. To our knowledge, these are partially created manually, which hampers its usage on *new* projects. It may thus be beneficial to investigate how this process can be automated.

References

[1] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 233–242.

[2] S. Counsell, G. Destefanis, X. Liu, S. Eldh, A. Ermedahl, and K. Andersson, "Comparing test and production code quality in a large commercial multicore system," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Aug 2016, pp. 86–91.

[3] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121217303060

[4] D. Spadini, M. Aniche, M.-A. Storey, M. Bruntink, and A. Bacchelli, "When testing meets code review: Why and how developers review tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, May 2018, pp. 677–687.

[5] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M. W. Mkaouer, A. Ouni, C. D. Newman, A. Ghallab, and S. Ludi, "Test smell detection tools: A systematic mapping study," in *Evaluation and Assessment in Software Engineering*, ser. EASE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 170–180. [Online]. Available: https://doi.org/10.1145/3463274.3463335

[6] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward, "On the introduction of automatic program repair in bloomberg," *IEEE Software*, vol. 38, no. 4, pp. 43–51, 2021.

[7] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Commun. ACM*, vol. 62, no. 12, p. 56–65, nov 2019. [Online]. Available: https://doi.org/10.1145/3318162

[8] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, p. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[9] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 672–681.

[10] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 157–166.

[11] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, Mar 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09750-5

[12] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 101–110.

[13] R. Kumar and A. V. Nori, "The economics of static analysis tools," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 707–710. [Online]. Available: https://doi.org/10.1145/2491411.2494574

[14] G. Horváth, P. Szécsi, Z. Gera, D. Krupp, and N. Pataki, "[engineering paper] challenges of implementing cross translation unit analysis in clang static analyzer," *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 171–176, 2018.

[15] B. Graham, "Enhancing code reviews with static analysis software tools," 2021. [Online]. Available: https://blogs.grammatech.com/enhancing-code-reviews-with-static-analysis

[16] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '77. New York, NY, USA: Association for Computing Machinery, 1977, p. 238–252. [Online]. Available: https://doi.org/10.1145/512950.512973

[17] R. Amadini, G. Gange, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Abstract Interpretation, Symbolic Execution and Constraints," in *Recent Developments in the Design and Implementation of Programming Languages*, ser. OpenAccess Series in Informatics (OASIcs), F. S. de Boer and J. Mauro, Eds., vol. 86. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 7:1–7:19. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/13229

[18] S. Gulwani and A. Tiwari, "Computing procedure summaries for interprocedural analysis," in *Proceedings of the 16th European Symposium on Programming*, ser. ESOP'07. Berlin, Heidelberg: Springer-Verlag, 2007, p. 253–267.

[19] E. Bodden, "The secret sauce in efficient and precise static analysis: The beauty of distributive, summary-based static analyses (and how to master them)," in *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ser. ISSTA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 85–93. [Online]. Available: https://doi.org/10.1145/3236454.3236500

[20] G. Horvath, P. Szecsi, Z. Gera, D. Krupp, and Z. Porkolab, "Cross translational unit analysis in clang static analyzer: Prototype and measurements," 2017. [Online]. Available: https://llvm.org/devmtg/2017-03//2017/02/20/accepted-sessions.html\#7

[21] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, Dec 2007.

[22] "Organizing and testing projects with the .net cli," 2022. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/core/tutorials/testing-with-cli

[23] S. Bechtold, S. Brannen, J. Link, M. Merdes, M. Philipp, J. de Rancourt, and C. Stein, "Junit 5 user guide - version 5.8.2," 2021. [Online]. Available: https://junit.org/junit5/docs/current/user-guide/\#writing-tests

[24] T. Muske and A. Serebrenik, "Techniques for efficient automated elimination of false positives," in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2020, pp. 259–263.

[25] M. Greiler, A. van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, March 2013, pp. 322–331.

[26] R. Ramler, M. Moser, and J. Pichler, "Automated static analysis of unit test code," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 2, 2016, pp. 25–28.

[27] SonarSource S.A., "Sonarsource static code analysis," (accessed 2021-06-21). [Online]. Available: https://rules.sonarsource.com/

[28] A. Alshammari, C. Morris, M. Hilton, and J. Bell, "Flakeflagger: Predicting flakiness without rerunning tests," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1572–1584.

[29] M. Monperrus, "The living review on automated program repair," HAL/archives-ouvertes.fr, Tech. Rep. hal-01956501, 2018.

[30] K. Etemadi, N. Harrand, S. Larsen, H. Adzemovic, H. L. Phu, A. Verma, F. Madeiral, D. Wikstrom, and M. Monperrus, "Sorald: Automatic patch suggestions for sonarqube static analysis violations," 2021. [Online]. Available: https://arxiv.org/abs/2103.12033

[31] K. Funk, "Clang-tidy, part 1: Modernize your source code using c++11/c++14," 2017. [Online]. Available: https://www.kdab.com/clang-tidy-part-1-modernize-source-code-using-c11c14/

[32] The LLVM Foundation, "Clang-Tidy," 2019, (accessed 2021.06.20). [Online]. Available: http://clang.llvm.org/extra/clang-tidy

[33] K. Beck, *Extreme Programming Explained*. Addison-Wesley, 2003.

[34] The LLVM Foundation, "Clang Static Analyzer," 2019, (accessed 2021.06.19). [Online]. Available: http://clang-analyzer.llvm.org

[35] ——, "Source Annotations," (accessed 2021.06.19). [Online]. Available: https://clang-analyzer.llvm.org/annotations.html

[36] D. Marjamäki, "Cppcheck: a tool for static c/c++ code analysis," 2013. [Online]. Available: http://cppcheck.sourceforge.net/

[37] M. Almarzouq, A. Alzaidan, and J. AlDallal, "Mining github for research and education: challenges and opportunities," *International Journal of Web Information Systems*, vol. ahead-of-print, 06 2020.

[38] The LLVM Foundation, "Cross Translation Unit (CTU) Analysis," (accessed 2022.01.19). [Online]. Available: https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html

[39] "Why google c++ testing framework?" 2022. [Online]. Available: https://chromium.googlesource.com/external/github.com/pwnall/googletest/+/refs/tags/release-1.8.0/googletest/docs/FAQ.md

[40] "SPA evaluation on test code." [Online]. Available: https://github.com/jean-malm-mdh/spa_comparison_public