Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Languages and Compilers

Enhancing pattern matching-based static analysis of C-family software projects with project-level knowledge

Supervisor:
Richárd Szalay
Ph.D. Student

Author:
Benedek Attila Bahrami
Computer Science B.Sc.

Budapest, 2022

This page should be the original Thesis Topic Declaration.

Contents

Chapter 1

Introduction

1.1 Motivation

There are programming languages, like Pascal, Ada or BASIC, that distinguish functions that do and do not have return values, the latter are known as *procedures*. C family languages do not. Pre-standardisation C language (also known as K&R C) [1] did not have `void` functions, instead an unspecified return value defaulted to `int`. This resulted in functions declared with `int` return value not returning anything and the supposed return value was unused on purpose. One of its consequences was for example, that for a while if you wrote a function, that was declared to return `int` without the return value, the compiler did not act on it. When writing code in C++ we often use functions from C, and a lot of standard library POSIX functions are C functions, taking `const char *`, `void *`, returning `int`.

1.2 Problem Statement

There are quite a few functions whose return value is often ignored, which could lead to potential bugs. Some examples:

- POSIX `read`: returns the number of bytes read; this return value can also indicate errors with it being -1.
- POSIX `scanf`: returns the number of items in the argument list successfully filled; also indicates errors with EOF return.
- (cstdio) `std::remove`: return indicates success or error of removal of a file.

- (algorithm) `std::remove`: Does not remove. It returns an iterator, and we still need to use container specific erase for all elements after this iterator.
- `std::remove_if`: Same as remove.
- container specific `erase`: Returns an iterator to the next element after the removed.
- container specific `insert`: Returns an iterator to the first of the new elements inserted.

Later the attribute `[[nodiscard]]` was introduced [2] to notify and give warnings to the user if the return value was unused in case of a function with this attribute, but in order to ask the compiler to give warnings on unchecked values, we would need permission to modify the library code. In case of external source code such as POSIX, STL or any third party project, we will not have permission to do so. However we still need to notify the user on the cases where they do not check non-void return value. This brings us back to static analysis.

1.3 Static analysis

Static analysis is a method to analyse the source code of software projects without performing a real execution of the application. It is widely used in industry [3] to find bugs and code smells during development, to aid in the prevention of bad code that misbehaves in production[4]. Among various methods, the most important techniques are the ones that are based on pattern matching on a syntactic representation of the software project.

Clang-Tidy is a declarative, object oriented, strong typed static analysis rule collection that is built upon the LLVM Compiler Infrastructure's C-family compiler, Clang. It performs pattern matching on Clang's *"Abstract Syntax Tree"* (AST) representation, and generating diagnostics based on which analysis modules, called *"checks"*, the user turns on. Pattern matching is an important tool in the industry as well. It can for example be used for type migration[5]. An example of an AST can be seen at fig. 1.1. We will address both the LLVM library and AST matchers in later chapters of the thesis.

Clang-Tidy's infrastructure is a powerful tool that can even be used to generate `FixIts`[6], but since a valid automatic fix does not exist for my checker's warnings, (since it is up to the user to decide whether or not the diagnosis is valid, or how they should acknowledge the result) it will not be used in this thesis.

```
TranslationUnitDecl
|-FunctionDecl <line:1:1, col:23> col:5 used foo 'int ()'
| `-CompoundStmt <col:11, col:23>
|   `-ReturnStmt <col:13, col:20>
|     `-IntegerLiteral <col:20> 'int' 1
`-FunctionDecl <line:3:1, line:5:1> line:3:5 main 'int ()'
  `-CompoundStmt <col:12, line:5:1>
    `-ReturnStmt <line:4:5, col:16>
      `-CallExpr <col:12, col:16> 'int'
        `-ImplicitCastExpr <col:12> 'int (*)()' <FunctionToPointerDecay>
          `-DeclRefExpr <col:12> 'int ()' lvalue Function 0x5629742c0820 'foo' 'int ()'
```

Figure 1.1: An example of an Abstract Syntax Tree of `return foo();`.

Let us imagine a checker, that keeps statistics on how many times a return value of a non-`void` function is used, or otherwise known as checked. This property is, as we previously stated important, because there exist a great amount of functions, whose return value should be checked in most situations but remain unchecked in quite a lot.

1.4 Infrastructure Limitations

Unfortunately, for programming languages in the C family, such as C++, the concept of "separate translation" causes issues for static analysis. As most static analysers are built upon compilers, and in C++, each compiler only sees the local information in the source file (also known as the *Translation Unit*) it is to compile or analyse (as opposed to project-level knowledge), crucial details might be hidden, which lowers, or in most cases, completely distorts the accuracy of the analysis. This means that the way our infrastructure works, a checker like this would be of very limited use.

A function can obviously exist outside of the translation unit of its declaration. If such analysis with our imagined checker is done separately on each translation unit, it is easy to see how that can affect the outcome. A function might be called 100 times and checked 95 times through. We would want to give warnings for the unchecked 5% of those calls, but if, for example, these are in a separate translation

unit, then the analysis would return with 0 checks out of 5 calls. We would not want to give warnings to a function that is unchecked in all 100% of its calls. The detail of the statistics, that it was only unchecked in 5% is lost, unless we use project level knowledge during our analysis. Consider another example, with code:

```
1  // First translation unit
2  std::vector<int> c{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 15};
3
4  auto iter = c.erase(c.begin());
5  iter = c.erase(iter + 2, iter + 5);
6  for (std::vector<int>::iterator it = c.begin(); it != c.end(); ) {
7      if (*it % 2 == 0)
8          it = c.erase(it);
9      else
10         ++it;
11 }
12
13 // Second translation unit
14
15 std::vector<int> c{0, 1, 2, 3, 4};
16 for (std::vector<int>::iterator it = c.begin(); it != c.end(); ++it
       ) {
17     if (*it % 2 == 0) {
18         c.erase(it);
19     }
20 }
```

Code 1.1: An example of the infrastructure's limitations.

Here, with separate analysis, we would get 100% checked in the first TU from 2 checked out of 2, and 0% in the second one from 0 checked out of 1. With project level knowledge, however we would get 66% from 2 checked out of 3. We used erase without a care towards its return value, and this could lead to potential bugs, but without the project level knowledge, however we can not diagnose it.

The separate analysis could also create false positive results. Imagine we have a function whose return value we could use but it is mostly optional. We could give our checker a treshold of percentage, to only diagnose unused values if we usually use them in most cases. Again, this means that with different translation units, we do not know how many times we actually ignored the value and can not use our treshold properly. This leads to false positive diagnosis.

Unfortunately, the current versions of Clang-Tidy checks can only access what is visible to the compiler, which is a local information. Several classes of security issues and bad coding patterns might be diagnosed if the implemented checks would be capable of creating percompilation knowledge, and reusing the full knowledge about the project during diagnosis. [7] The work of the thesis is to enhance Clang-Tidy on the infrastructure level to support multi-pass analyses in a generic manner, by utilizing the ideas similar to that of MapReduce [8].

This is achieved by allowing individual checks to store check-specific data on a thread-safe location. A subsequent execution of the analysis will be able to do the pattern matching fine-tuned with the data stored in the previous step also available. To prove the usability of the solution, a new safety and security related check, currently not provided by Clang-Tidy, will be developed utilizing the new infrastructure created in this work. In the end, the results of the thesis will allow the international community behind LLVM to develop and make available a wider potential of checks, as we are planning to upstream the entire project into LLVM.

- D124446: Add the misc-discarded-return-value check [9]
- D124447: Add infrastructure support for running on project-level information [10]
- D124448: Add project-level analysis support to misc-discarded-return-value [11]

1.5 Related Works

The topic of cross translation unit analysis shares other different solutions and implementations as well, one great example is Clang CTU [12]. As previously stated, the infrastructure enhancement follows the philosophy of MapReduce [8]. There exists a static analysis rule in Coverity Scan static analysis tool, that is similar to our checker [13].

1.6 Thesis layout

After the Introduction, in chapter 2, the User Guide will have instructions on how to download, compile and set up the static analysis tool, and how to run it on

a C++ project. In chapter 3, the Developer Guide will explain how the checker, and the infrastructure itself works, in detail.

Chapter 2

User documentation

Both the changes in the Clang-Tidy infrastructure and our new checker obviously focuses heavily on LLVM's Clang-Tidy. Clang-Tidy is a clang-based C++ "linter" tool. Its purpose is to provide an extensible framework for diagnosing and fixing typical programming errors, like style violations, interface misuse, or bugs that can be deduced via static analysis. Clang-Tidy is modular and provides a convenient interface for writing new checks.

This tool can be found in the LLVM project repository.

2.1 Install guide

2.1.1 System Requirements

Table 2.1 shows the system requirements and supported compilers for building LLVM. The checkers were developed with Ubuntu 20.04 and tested on Ubuntu 18.04, and WSL Ubuntu 20.04.

Building and using LLVM's Clang-Tidy takes a lot of time on weaker computers. The minimum recommended memory size for building is 16 GB, the optimal amount is 64 GB of memory.

Operating System	Processor Architecture	Compiler
Linux	x861	gcc, clang
Linux	amd64	gcc, clang
Linux	arm	gcc, clang
Linux	Mips	gcc, clang
Linux	PowerPC	gcc, clang
Solaris	V9	gcc
FreeBSD	x861	gcc, clang
FreeBSD	amd64	gcc, clang
NetBSD	x861	gcc, clang
NetBSD	amd64	gcc, clang
macOS2	PowerPC	gcc
macOS	x86	gcc, clang
Cygwin	x86	gcc
Windows	x86	Visual Studio
Windows64	x86-64	Visual Studio

Table 2.1: System requirements and supported compilers for building LLVM.

Software requirements include (at least) GCC version 7.1.0, CMake version 3.13.4, Python version 3.6 and GNU Make version 3.79.

2.1.2 Building from source

These commands will compile LLVM from source. The building process with parameters can be found on the README.md of LLVM project Github repository[1], or the Getting Started[2] page of Clang documentation. These are the commands I used for the compilation.

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project
mkdir Build

# cmake -S llvm -B build -G <generator> [options]
cd Build/
cmake \
  -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
  -DLLVM_ENABLE_PROJECTS="llvm;clang;clang-tools-extra" \
  -DLLVM_TARGETS_TO_BUILD="X86" \
  -DLLVM_APPEND_VC_REV=OFF \
  -DLLVM_ENABLE_BINDINGS=OFF \
```

[1] https://github.com/llvm/llvm-project#readme, accessed 2022. 05. 03.
[2] https://clang.llvm.org/get_started.html, accessed 2022. 05. 03.

```
    -DLLVM_USE_RELATIVE_PATHS_IN_FILES=OFF \
    -DBUILD_SHARED_LIBS=ON \
    -DLLVM_USE_LINKER="lld" \
    -DLLVM_PARALLEL_LINK_JOBS=3 \
    -DCMAKE_BUILD_TYPE=Release \
    -DLLVM_ENABLE_DUMP=ON \
    -DLLVM_ENABLE_ASSERTIONS=ON \
    -G Ninja \
    ../llvm

# cmake --build build [-- [options] <target>] or your build
    system specified above directly.
ninja -j12 clang-tidy llvm-symbolizer
```

Explanation for some flags: at `DLLVM_USE_LINKER` we can change the linker we are using, either Gold or LLD (Linker for LLVM). The latter needs to be installed. `DLLVM_PARALLEL_LINK_JOBS` and -j at Ninja sets the CPU capacity. The recommended amount for `LINK_JOBS` is one quarter of the amount of cores, and Ninja job amount should be $cores - 2$. I used these commands on a server with 32 GB memory and 14 CPU cores.

2.2 Running by Translation Units

You can give Clang-Tidy multiple translation units to run on, and it will give you diagnosis separately for each one. You run it by using `clang-tidy -checks='-*,misc-discarded-return-value' -p ./Build a/main.cpp b/main.cpp`, where the "checks" first disables all checkers with -*, then enables our checker, the flag "p" gets the build path and finally we give the paths to our code. Here we are getting two separate diagnoses for our two separate files or translation units.

```
/home/bahramib/MyFolder/TestFolder/a/main.cpp:12:9: \
warning: return value of 'maybe_check_this' is used \
in most calls, but not in this one [misc-discarded-return-value]
    MyClass::maybe_check_this();
    ^
/home/bahramib/MyFolder/TestFolder/a/main.cpp:12:9: note: \
value consumed or checked in 75% (3 out of 4) of cases
```

Code 2.1: Diagnosis output without project level knowledge.

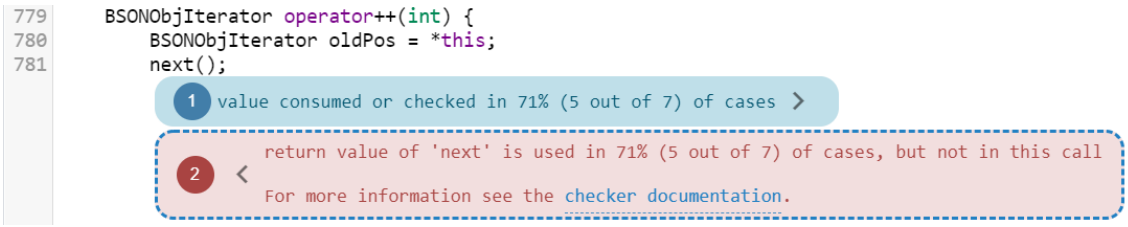


Figure 2.1: A report of the diagnosis of MongoDB's source on CodeChecker with 50% treshold on a single TU.

Clang-Tidy is supported by CodeChecker. [14]

2.2.1 Configuring Treshold

As previously descussed, the ratio of the amount of checked function calls and all the function calls is very important for the diagnosis, since we do not necessarily want to be notified of all the unused calls, but let us say if a function's return value is used in at least 65% of all calls, we might want to be warned about the missing unchecked 35%.

The checker has an option for this called the `treshold`, that is the percentage above which we would like to be warned, but not under. You can configure the treshold, and any Clang-Tidy checker option with Tidy's `-config` flag. Here you will have to type the name of the checker and its option member, with the desired value as a `key-value pair`. This is how it looks with 50%: `-config='{CheckOptions: [{key : misc-discarded-return-value.ConsumeThreshold, value: 50}]}'`. With this option the full command looks like this: `clang-tidy -checks='-*,misc-discarded-return-value'-config='{CheckOptions: [{key: misc-discarded-return-value.ConsumeThreshold , value: 50}]}' -p ./Build a/main.cpp b/main.cpp`.

2.2.2 Supressing Warnings

Let us review what is wrong with our code in listing 2.2. In function foo, we forget to use our parameter p, and we do not check the return value when calling foo. These triggers one warning for each mistake as seen in listing fig. 2.2.

```
1 [[nodiscard]] int foo(int p) {
2   return 1;
3 }
4 int main(int argc, char**) {
5   foo(argc);
```

```
}
```

Code 2.2: An example of both unused parameter and ignored return value with nodiscard.

```
<source>:1:27: warning: unused parameter 'p' [-Wunused-parameter]
[[nodiscard]] int foo(int p) {
                          ^
<source>:5:3: warning: ignoring return value of function declared with 'nodiscard' attribute [-Wunused-result]
  foo(argc);
  ^~~ ~~~~
```

Figure 2.2: The compiler warnings.

If we do not want to fix these issues, but rather silence our warnings, we can simply write `(void)` before the call to cast it into `void` and trick the compiler into thinking that we did check the return value. Similarly, we could cast our parameter p into void as a distinct statement to make it seem like we used it. On code listing 2.3 we can see the suppressed version that does not yield warnings.

```
[[nodiscard]] int foo(int p) {
  (void)p;
  return 1;
}
int main(int argc, char**) {
 (void)foo(argc);
}
```

Code 2.3: The same example, now with suppressed warnings.

The suppression of our new checker warnings follows the exact same convention. Casting into void acts as usage of the return value and thus will not give warnings, despite not actually checking it.

Both the philosophy and practice of silencing a single case of an unchecked function call is essentially the same as silencing the warning of `[[nodiscard]]` or an unused parameter. Simply write `(void)` before the call, to cast it into void, tricking the compiler and the checker into thinking it is used, but not affecting anything in reality. If we got a warning on the line `foo();`, from the checker or `[[nodiscard]]`, we will get neither if we change it to `(void)foo();`.

2.3 Multiple Phase Version

The updated infrastructure contains two new flags for running Clang-Tidy, `multipass-phase` and `multipass-dir`. `Multipass-phase` is an `enum` flag, that has three values, `"collect"`, `"compact"` and `"diagnose"` with the latter as default. `Multipass-dir` needs a path to a directory where the checkers that support the collect feature can dump their collection datas that they are going to compact and use later.

2.3.1 Collect

Collect phase, as the name suggest, will have the checkers collect data on each translation unit and write them into unique YAML files to later reuse this data. This is how you normally run collect phase on the desired files:

```
clang-tidy \
-checks='-*,misc-discarded-return-value' \
--multipass-phase=collect \
--multipass-dir='MyCollectionDirectory' \
-p ./Build \
a/main.cpp b/main.cpp
```

What Discarded Return Value checker does in this phase, is count the amount the declared non-void functions were called, and count the amount that these function's return values were checked. After finishing counting in one translation unit, it writes the collected numbers and function names into a YAML file as a struct.

After collecting, you do not get any diagnosis or output text, but the desired amount of (in this case 2) YAML files are going to be generated.

```
bahramib@cc:~/MyFolder/TestFolder/MyCollectionDirectory$ ll
total 8
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:00 misc-discarded-
    return-value.main.cpp.12949585208029997868.yaml
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:00 misc-discarded-
    return-value.main.cpp.4924802982073527590.yaml
```

Code 2.4: The YAML files containing the collection data.

```
# First TU
---
```

```
- ID:               'c:@S@MyClass@F@check_that#S'
  Consumed:         3
  Total:            3
- ID:               'c:@S@MyClass@F@maybe_check_this#S'
  Consumed:         0
  Total:            3
...
# Second TU
---
- ID:               'c:@S@MyClass@F@check_that#S'
  Consumed:         1
  Total:            3
- ID:               'c:@S@MyClass@F@maybe_check_this#S'
  Consumed:         3
  Total:            4
...
```

Code 2.5: Contents of the collection files.

2.3.2 Compact

Compact phase will iterate through the collected data per checker and have the checkers read, use and compact all the data collected into one YAML file. Flags aside from checks, multipass-phase and multipass-dir have no effect.

```
clang-tidy \
-checks='-*,misc-discarded-return-value' \
--multipass-phase=compact \
--multipass-dir='MyCollectionDirectory' \
```

This checker reads the data on each function back and constructs new data similar to the previous ones. If one function is called in multiple translation units, then it simply adds the numbers from the TU's YAML file to the new structure. After it is finished, the data is written into a single YAML file.

This phase does not write anything on standard output either, but will construct the compacted YAMLs per checker.

```
bahramib@cc:~/MyFolder/TestFolder/MyCollectionDirectory$ ll
total 12
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:00 misc-discarded-
    return-value.main.cpp.12949585208029997868.yaml
```

```
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:00 misc-discarded-
    return-value.main.cpp.4924802982073527590.yaml
-rw-r--r-- 1 bahramib bahramib 196 Apr 30 13:06 misc-discarded-
    return-value.yaml
```

Code 2.6: The new file containing the collected data.

```
---
- ID:              'c:@S@MyClass@F@check_that#S'
  Consumed:        4
  Total:           6
- ID:              'c:@S@MyClass@F@maybe_check_this#S'
  Consumed:        3
  Total:           7
...
```

Code 2.7: Contents of the compacted file.

2.3.3 Diagnose

For backwards compatibility, the default diagnose phase, will do exactly what the non-multiple phase Clang-Tidy did, give diagnoses for each translation unit separately, as demonstrated in section section 2.2, if compact has not happened for a checker. Otherwise that checker will read and use the data compacted in the respective YAML file, and give its diagnosis calculated with project level knowledge.

```
clang-tidy \
-checks='-*,misc-discarded-return-value' \
--multipass-phase=diagnose \
--multipass-dir='MyCollectionDirectory' \
-p ./Build \
a/main.cpp b/main.cpp
```

My checker reads in the compacted data with the call and check amounts for each function and simply desides if diagnosis is needed or not for each unchecked return value in the current translation unit. The output is obviously the diagnosis.

```
/home/bahramib/MyFolder/TestFolder/a/main.cpp:12:9: \
warning: return value of 'check_that' is used \
in most calls, but noti in this one [misc-discarded-return-value]
MyClass::check_that();
```

```
    ^
/home/bahramib/MyFolder/TestFolder/a/main.cpp:12:9: note: \
value consumed or checked in 66% (4 out of 6) of cases

/home/bahramib/MyFolder/TestFolder/a/main.cpp:15:5: \
warning: return value of 'check_that' is used \
in most calls, but not in this one [misc-discarded-return-value]
MyClass::check_that();
^
/home/bahramib/MyFolder/TestFolder/a/main.cpp:15:5: note: \
value consumed or checked in 66% (4 out of 6) of cases
```

Code 2.8: Diagnosis output with project level knowledge.

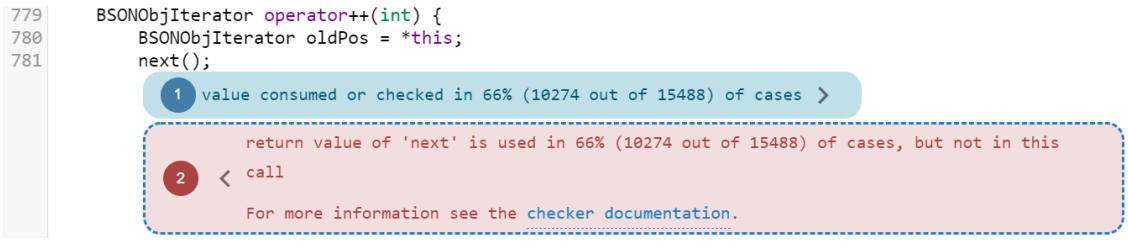


Figure 2.3: A report of the diagnosis of MongoDB's source on CodeChecker with 50% treshold and project level knowledge.

We can clearly conclude, that the multipass diagnosis resulted in different warnings. Figure fig. 2.3 differs from fig. 2.1 in only the percentage, but with a different treshold that could lead to different results, to either false positives, or hidden true positives. Our own example, however shows both, since we had a call that used to give warning when diagnosed separately which with project level knowledge does not, and one that did not give any warnings, but now it gives two, because it passed the treshold.

In listing 1.1 we talked about our code getting no warnings without project level knowledge. After phases collect and compact, however our diagnosis will result in the following:

```
/home/bahramib/MyFolder/TestFolder/b/vec.cpp:7:15: \
warning: return value of 'erase' is used \
in most calls, but not in this one [misc-discarded-return-value]
            c.erase(it);
              ^
/home/bahramib/MyFolder/TestFolder/b/vec.cpp:7:15: note: \
```

```
7 value consumed or checked in 66% (2 out of 3) of cases
```

We can clearly see, that our diagnosis gave us the real results this time.

Chapter 3

Developer documentation

This chapter will cover the process of developing a Clang-Tidy checker, and the infrastructure that it is built on. The developer should have an advanced knowledge of C++ and strong typing, minimal user experience with Linux and Clang-Tidy, and familiarity with declarative programming paradigm.

The work of the thesis and all the code I have written is a contribution and enhancement to the originally existing library of LLVM Compiler Infrastructure. I have attached the patch file that contains all the modifications to the original code with the thesis.

3.1 Infrastructure

The enhancement of the infrastructure focused on implementing the three distinct phases previously discussed in chapter 2 for the checkers to distinguish and the developers to use. Figure fig. 3.1 and fig. 3.2 demonstrates difference between the old infrastructure (`"single"`) and the new `"multi"`. Note that if the aforementioned compacting has not happened for a checker, the third phase (diagnosis) will perform the "single" path, as it is the default mode of Clang-Tidy for both backwards compatibility, and intentional usage.

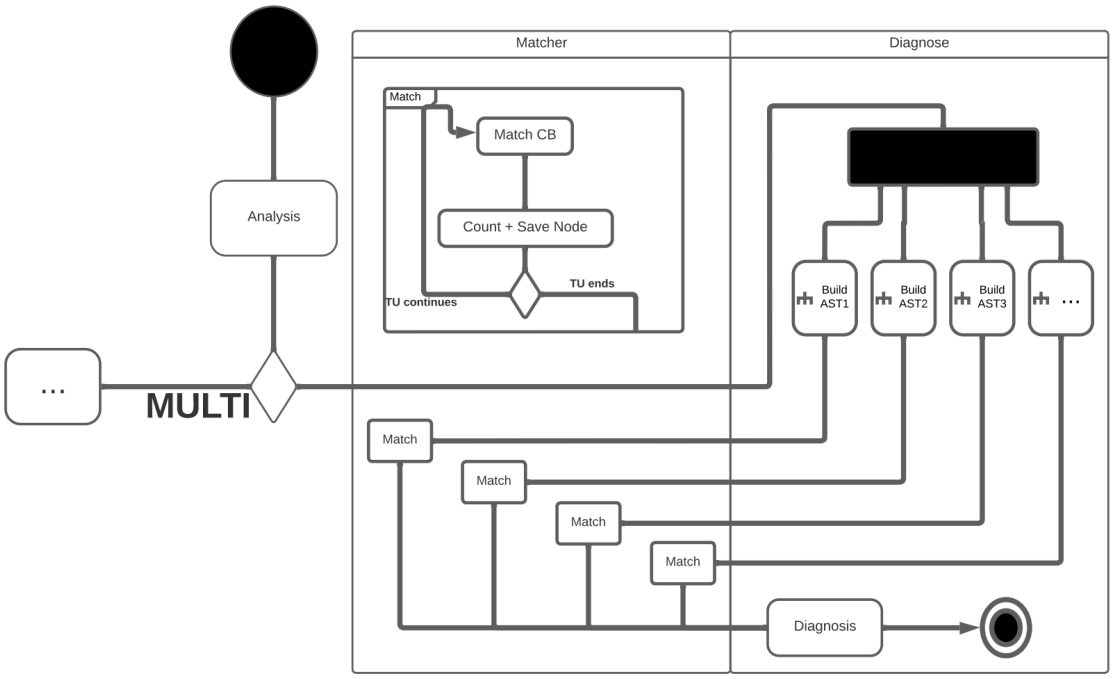


Figure 3.1: Activity diagram of Clang-Tidy infrastructure in "single" mode.

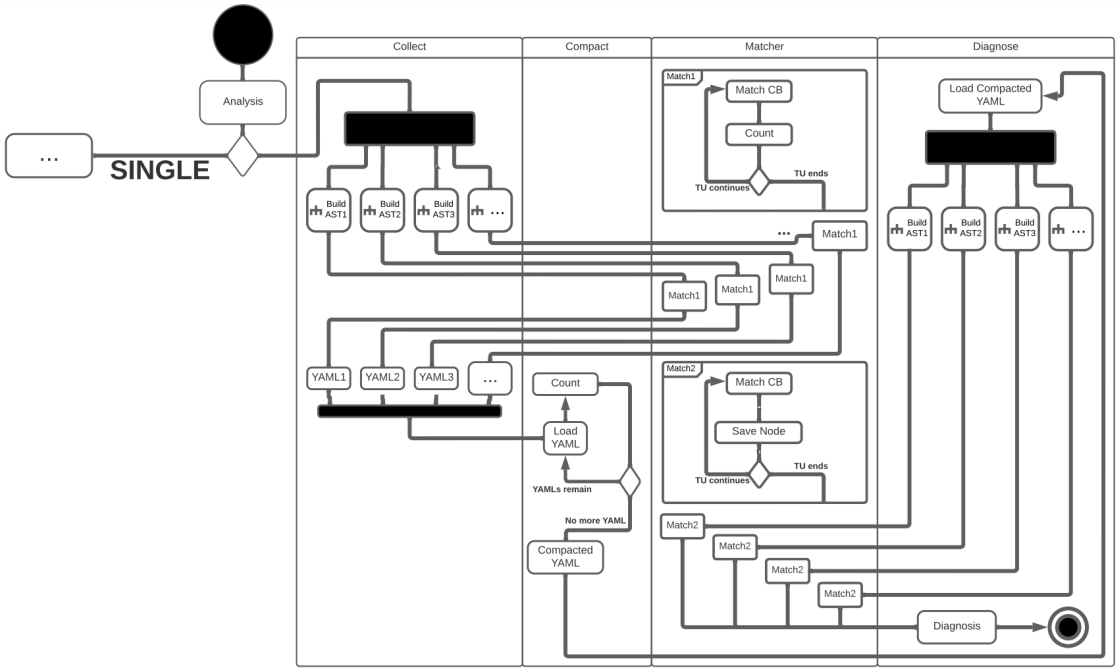


Figure 3.2: Activity diagram of Clang-Tidy infrastructure in "multi" mode.

The new flags I implementedare an `enum` flag for the current phase that we are in, and a regular `string` for the collection directory path. If the directory does not currently extists on the given path, the infrastructure will create one.

The base class of the checkers, `ClangTidyCheck` already has `virtual` functions for the checkers to override, with some being optional and some not. There are

functions for `language support`, registering `preprocessor callbacks`, registering `matchers`, checking matched `AST nodes`, and storing checker `options`. There are two additional functions from the base of `ClangTidyCheck`, `MatchCallback`, and these are `onStartOfTranslationUnit and onEndOfTranslationUnit`, to be able to do separate commands before and after the checker's actual work is done.

```
/// Override this to register ``PPCallbacks`` in the preprocessor.
virtual void registerPPCallbacks(const SourceManager &SM,
    Preprocessor *PP,
  Preprocessor *ModuleExpanderPP) {}

/// Override this to register AST matchers with Finder.
virtual void registerMatchers(ast_matchers::MatchFinder *Finder) {}

/// ``ClangTidyChecks`` that register ASTMatchers should do the
    actual
/// work in here.
virtual void check(const ast_matchers::MatchFinder::MatchResult &
    Result) {}

/// Should store all options supported by this check with their
/// current values or default values for options that haven't been
    overridden.
virtual void storeOptions(ClangTidyOptions::OptionMap &Options) {}
```

Code 3.1: Virtual functions from ClangTidyCheck's header.

```
/// Called at the start of each translation unit.
/// Optionally override to do per translation unit tasks.
virtual void onStartOfTranslationUnit() {}

/// Called at the end of each translation unit.
/// Optionally override to do per translation unit tasks.
virtual void onEndOfTranslationUnit() {}
```

Code 3.2: Virtual functions from MatchCallback's header.

Originally, when a `callback` happened (which we will address later in section 3.2) `ClangTidyChecker` simply called the derived `class`'s `check` function.

```
void ClangTidyCheck::run(const ast_matchers::MatchFinder::
    MatchResult &Result) {
  check(Result);
```

```
}
```

Code 3.3: The old infrastructure's way of calling check.

Now the checkers have three new `virtual void` member functions to override, `collect`, `postCollect` and `compact`. It is important to note that this override is optional, in a sense, that some checkers can not be improved with and do not need project level knowledge. If one however decides to use the new multipass infrastructure with one's checker, then none of these overrides are optional.

`ClangTidyChecker` has a private `ClangTidyContext` member called `Context`. This contains information about the current phase, the directory path and the phase `enum` as well. This was used in the newly implemented member functions of the `ClangTidyChecker` parent `class`, and with protected getter functions, the derived checker can use some information from this context.

As we can see in listing 3.3, this function only called the checker's `check`, but now it distinguishes two out of our 3 separate phases, diagnose and collect. In diagnose phase, it still calls for `check`, but in collect phase it obviously calls for the checker's separate `collect` function.

```
void ClangTidyCheck::run(const ast_matchers::MatchFinder::
    MatchResult &Result) {
  switch (getPhase()) {
  case MultipassProjectPhase::Diagnose:
    check(Result);
    break;
  case MultipassProjectPhase::Collect:
    collect(Result);
    break;
  case MultipassProjectPhase::Compact:
    llvm_unreachable("AST Matchers should not have run in compact
        mode.");
  }
}
```

Code 3.4: Run function distinguishing Diagnose and Collect phase.

After all the collection of each checker has been done on a translation units, the destructor of the `class` that creates the checkers will call for the `runPostCollect` function of the base `class`, which as the name suggests, runs the `postCollect` function of the derived checker.

On compact phase, the infrastructure does not need the translation units at all, so there are no checkers instantiated during the compilation, since this phase excludes compilation. Instead the checker creation is now refactored into a function and that is called in a separate path of the code, where we only create the checkers and call compact on each of them once.

In the following subsections I will explain how the `virtual` functions should and how the member functions do work.

3.1.1 Collect Function

`Collect` recieves the same parameters as check does, a `MatchResult` reference. In this function the checker should do something similar to itsoriginal check function. Get information out of the translation unit, and store it in a data structure of choise that is fit for the checker's logic.

3.1.2 PostCollect Function

`PostCollect` recieves a `StringRef`, the name of the output YAML file. This function a checker should take all the data collected from the translation unit, from the data structure, and write it in a YAML file with the function parameter as filename.

3.1.3 Compact Function

`Compact` has two parameters. One is a `StringRef` that, similarly to `postCollect`, contains the name of the output file. The other is a `vector` of `strings` that contains all the YAML files in our multipass directory, that the current checker created in the collect phase and wrote the collection data into for each translation unit. In this phase, the checker should collect all the data from these YAML files and compact them into a single data structure. After this step, it should write it into the YAML with the name of the output filename parameter.

3.1.4 Additional Functions

There are newly implemented protected functions to use aside from these `virtual` ones:

- `getCollectPath`: returns a `string` to the path where the current check should write collected data to.
- `getCompactedDataPath`: returns a `StringRef`, the name of the file where the current check should write or read compacted data to/from.
- `getPhase`: returns the enum representing the current phase

3.2 Discarded Return Value Check

In section section 1.2 we justified the existence of a checker which focuses on detecting potential bugs and code smells and giving warnings to function calls where the return value is ignored or unchecked if we deem it necessary by analizing the ratio of the amount of unchecked calls and all calls for the same function.

3.2.1 Abstract Syntax Trees

Before making a checker, first we should know about how checkers work. Clang-Tidy checkers perform `pattern matching` on Clang's `object oriented` and `strong typed` representation of the `abstract syntax tree` of the source code. ASTs represent the syntactic structure of a code as seen in fig. 1.1.

In this figure we have 2 `function declarations`, one of which is our main. We can see that our other function is called `foo`, that takes `no parameters` and returns an `int`. In the `body` we have a statement, more specifically a return statement. The subject of the return statement is a stimple `integer literal`, with the value of 1. In main, we only have another return statement, that has a `call expression` of a function that has a `declaration reference expression` that refers to the declaration of our function `foo`.

Obviously ASTs get increasingly more complex and harder to read like this. Another example can be seen on figure fig. 3.3.

```
TranslationUnitDecl
`-FunctionDecl <line:3:1, line:7:1> line:3:5 main 'int ()'
  `-CompoundStmt <col:12, line:7:1>
    `-ForStmt <line:4:5, line:6:15>
      |-DeclStmt <line:4:10, col:19>
      | `-VarDecl <col:10, col:18> col:14 used n 'int' cinit
      |   `-IntegerLiteral <col:18> 'int' 0
      |-<<<NULL>>>
      |-BinaryOperator <col:21, col:26> 'bool' '<='
      | |-ImplicitCastExpr <col:21> 'int' <LValueToRValue>
      | | `-DeclRefExpr <col:21> 'int' lvalue Var 0x5574beea9ad8 'n' 'int'
      | `-IntegerLiteral <col:26> 'int' 10
      |-UnaryOperator <col:30, col:32> 'int' lvalue prefix '++'
      | `-DeclRefExpr <col:32> 'int' lvalue Var 0x5574beea9ad8 'n' 'int'
      `-IfStmt <line:5:9, line:6:15>
        |-BinaryOperator <line:5:13, col:22> 'bool' '=='
        | |-BinaryOperator <col:13, col:17> 'int' '%'
        | | |-ImplicitCastExpr <col:13> 'int' <LValueToRValue>
        | | | `-DeclRefExpr <col:13> 'int' lvalue Var 0x5574beea9ad8 'n' 'int'
        | | `-IntegerLiteral <col:17> 'int' 2
        | `-IntegerLiteral <col:22> 'int' 0
        `-UnaryOperator <line:6:13, col:15> 'int' lvalue prefix '++'
          `-DeclRefExpr <col:15> 'int' lvalue Var 0x5574beea9ad8 'n' 'int'
```

Figure 3.3: The AST of listing 3.5

```
1 int main() {
2   for (int n = 0; n <= 10; ++n)
3     if (n % 2 == 0)
4       ++n;
5 }
```

Code 3.5: The code of figure fig. 3.3.

3.2.2 Pattern Matching

Clang-Tidy checkers have `matchers` that you can set up in the checker's overridden function `registerMatchers`. This has its matcher, a `MatchFinder` as the parameter, and in this function we have to use its `addMatcher` member function to add new patterns for it to match, and also to `bind` a string literal to the matches for later usage. If the matcher finds a match, a `callback` action is executed.

In order to utilize the matches and the callbacks we have to know how to match nodes on an AST. Clang-Tidy has built in matchers, that we can see in the AST Matcher Reference [15]. This tells us which matcher accepts which matcher or

matchers as parameter, and what their return type is. It also gives us examples of usage. Theoretically, there are 3 distinct base types for AST nodes. `Declaration`, `Statement` and Type. In practice Clang-Tidy has more distinct bases in its AST representation, but for now let us view a a simple example of matching from `Compiler Explorer`[1]:

```
1 int foo() { return 1; }
2
3 int main() {
4   foo();
5   int n = foo();
6   return foo();
7 }
```

Code 3.6: A very simple code for matching function calls.

```
TranslationUnitDecl
|-FunctionDecl <line:1:1, col:23> col:5 used foo 'int ()'
| `-CompoundStmt <col:11, col:23>
|   `-ReturnStmt <col:13, col:20>
|     `-IntegerLiteral <col:20> 'int' 1
`-FunctionDecl <line:3:1, line:7:1> line:3:5 main 'int ()'
  `-CompoundStmt <col:12, line:7:1>
    |-CallExpr <line:4:5, col:9> 'int'
    | `-ImplicitCastExpr <col:5> 'int (*)()' <FunctionToPointerDecay>
    |   `-DeclRefExpr <col:5> 'int ()' lvalue Function 0x564ea6f9a820 'foo' 'int ()'
    |-DeclStmt <line:5:5, col:18>
    | `-VarDecl <col:5, col:17> col:9 n 'int' cinit
    |   `-CallExpr <col:13, col:17> 'int'
    |     `-ImplicitCastExpr <col:13> 'int (*)()' <FunctionToPointerDecay>
    |       `-DeclRefExpr <col:13> 'int ()' lvalue Function 0x564ea6f9a820 'foo' 'int ()'
    `-ReturnStmt <line:6:5, col:16>
      `-CallExpr <col:12, col:16> 'int'
        `-ImplicitCastExpr <col:12> 'int (*)()' <FunctionToPointerDecay>
          `-DeclRefExpr <col:12> 'int ()' lvalue Function 0x564ea6f9a820 'foo' 'int ()'
```

Figure 3.4: The AST of listing 3.6.

Here we can have 1 function declaration of the same function, and 3 call expression, the first one in as a regular statement, the second one in a variable declaration, and the third one in a return statement. Let us build a matcher for calls that a checker like mine should be concerned about. First we want to match for function calls, whose declaration reference refers to a function declaration. If we use `match callExpr(callee(declRefExpr(to(functionDecl().bind("definition")))))` `.bind("call")` we get 3 matches and 6 binds. It matches all 3 calls of `foo` bound to

[1] `https://godbolt.org/z/ncToe315q`, accessed 2022. 05. 14.

the literal "call", and we also get 3 binds, all for the first line, the FunctionDecl of foo, bound to "definition".

We can also save the matcher bound to "definition" as a variable for easier usage and simplicity, if we wanted to. With `let Def functionDecl().bind("definition")` it will be saved as Def, and with `let Call callExpr(callee(declRefExpr(to(Def)))).bind("call")`, where we reuse our Def, we save our Call for later use. Now we only have to match for Call with `match Call` to get the same results.

We could improve it this matcher for the checker, with not matching functions declared with `void` as return value. In Clang-Tidy the "saved" matcher will be `static const auto Call = callExpr(hasDeclaration(functionDecl(unless(returns(voidType())))));`. In reality, the checker will need more specification in order to work properly, but this distinction is important as well.

There are separate matchers in static variables for any kind of usage of a return value, and then these are added in a single `addMatcher` call of the matcher. Let us go through some of these matchers.

Call

Our actual specific call expression can be seen in listing 3.7. Since this is the base of all other matchers that we use, I will explain what we need to match for, in detail, starting with the bind. The `static constexpr` that Call is bound to is an `llvm::StringLiteral` type, that wraps "call". We want to match for function calls, but not all function calls. First we will sort out all functions whose declaration is a function declaration to at least either a `void` function, because they do not return a value to be checked or ignored, and a function with `[[nodiscard]]`, since the compiler will already warn the user if needed. This is what we achive with lines 2-3. Second we want to sort out any operator calls except for the following: `()` or `[]` operators (this is done by lines 4-5), and `*` or `&` operators, but since those operators have multiple uses and we want to include a specific use only, we only need these if they take only one arguement, also known as the `dereference` and the `reference` operator (which is achieved by lines 6-7). All of this specification can be seen in figure fig. 3.5.

```
1 static const auto Call =
2   callExpr(hasDeclaration(functionDecl(unless(anyOf(
3         returns(voidType()), hasAttr(attr::WarnUnusedResult))))),
```

```
4       unless(cxxOperatorCallExpr(
5         unless(anyOf(hasAnyOverloadedOperatorName("()", "[]"),
6                allOf(hasAnyOverloadedOperatorName("*", "&"),
7                  argumentCountIs(1)))))))
8     .bind(::Call);
```

Code 3.7: The matcher for our desired call expression.

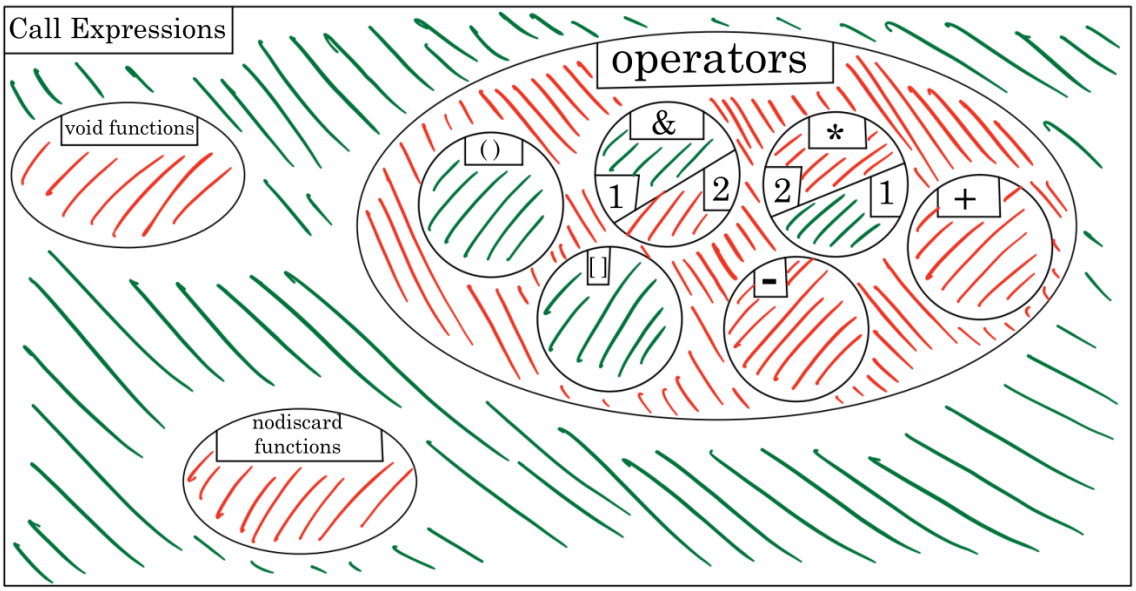


Figure 3.5: A Venn diagram of what Call matches, where 1 and 2 indicates arity

While

In listing 3.8 we match for either a `while statement` or a `do statement` that uses our call in their condition, and then since the matcher `anyOf` does not automatically cast its return type to that of a statement, we do it manually with `stmt`.

```
1 static const auto While =
2   stmt(anyOf(whileStmt(hasCondition(Call)), doStmt(hasCondition(
      Call))));
```

Code 3.8: The matcher for usage in while expression.

Return

In listing 3.9 we simply match if a return statement uses our function as its return value.

```
1 static const auto Return = returnStmt(hasReturnValue(Call));
```

Code 3.9: The matcher for the usage in return statement.

For

In listing 3.10 we match for any and all of the following: if the `initialization`, the `condition`, or the `incrementation` of the for loop, or if a `range-based` for loop, or any descendants of either of these uses the return value of our call.

```
static const auto For = anyOf(
  forStmt(eachOf(hasLoopInit(findAll(Call)), hasCondition(findAll(
     Call)),
          hasIncrement(findAll(Call)))),
  cxxForRangeStmt(hasRangeInit(findAll(Call))));
```

Code 3.10: The matcher for the usage in for statement.

We can also create new matchers, should we need one that is not yet in the library. It provides `macros` that can be used to implement matchers with custom logic. For example in listing 3.11 the macro defines a single-parameter function called `hasAssertExpr` that takes a `Matcher<Expr>` called `InnerMatcher` and returns a `Matcher<StaticAssertDecl>` object. What it does is very simple. It passes the matcher into the assertion expression of assert, meaning if the inner matcher matches, the entire assertion will too.

```
AST_MATCHER_P(StaticAssertDecl, hasAssertExpr,
              ast_matchers::internal::Matcher<Expr>, InnerMatcher)
                 {
  return InnerMatcher.matches(*Node.getAssertExpr(), Finder,
     Builder);
}

static const auto StaticAssert = staticAssertDecl(hasAssertExpr(
    Call));
```

Code 3.11: Custom logic and usage of a new matcher called `hasAssertExpr`.

3.2.3 Checker Logic

Before going into how my checker works, we should go through the member functions of it. The checker obviously overrides all non-optional and some optional

`virtual` functions as seen in listing 3.12. In listing 3.13 there are three additional member functions. These contain the logic of the checker and how it shapes its data structure according to the current phase.

```
void registerMatchers(ast_matchers::MatchFinder *Finder) override;
void storeOptions(ClangTidyOptions::OptionMap &Opts) override;
void onStartOfTranslationUnit() override;
void onEndOfTranslationUnit() override;
void collect(const ast_matchers::MatchFinder::MatchResult &Result)
    override;
void postCollect(StringRef OutputFile) override;
void compact(const std::vector<std::string> &PerTuCollectedData,
       StringRef OutputFile) override;
void check(const ast_matchers::MatchFinder::MatchResult &Result)
    override;
```

Code 3.12: Overridden virtual functions in DiscardedReturnValueCheck's header.

```
void matchResult(const ast_matchers::MatchFinder::MatchResult &
    Result,
                 bool ShouldCount);
void registerCall(const CallExpr *CE, const FunctionDecl *FD,
                  bool IncrementCounters, const void *
                     ConsumingContext);
void diagnose(const Function &F);
```

Code 3.13: Private member functions in DiscardedReturnValueCheck's header.

For the data structure itself, The checker uses a `struct` representing the functions with their tracked amount of calls and checks (and its member function calculating the ratio of these in percentage), a `StringMap` of functions to store their name and their corresponding struct object, a `DenseMap` of `{FunctionDecl - string` pairs, a `DenseMap` of `CallExpr - SmallVector` pairs to prevent multiple callbacks on a single call expression being processed the twice. It also uses a `uint8_t` to save the the treshold option, and an `Optional<bool>` to track whether or not compacting has been done on the translation units.

```
struct Function {
  std::size_t ConsumedCalls;
  std::size_t TotalCalls;
  const FunctionDecl *FD;
  llvm::SmallPtrSet<const CallExpr *, 32> DiscardedCEs;
```

```
6
7   std::uint8_t ratio() const;
8 };
9   using FunctionMapTy = llvm::StringMap<Function>;
10
11  const std::uint8_t ConsumeThreshold;
12  Optional<bool> CacheProjectDataLoadedSuccessfully;
13
14  llvm::DenseMap<const CallExpr *, SmallVector<const void *, 2>>
        ConsumedCalls;
15  llvm::DenseMap<const FunctionDecl *, std::string> FunctionIDs;
16  FunctionMapTy CallMap;
```

Code 3.14: Members of the data structure.

My checker works with a very simple philosophy. After all the possible usages of a return value have been refactored into these matchers, we put all of them into one call of `addMatcher` and bind them to `"consume"`. In reality, the usages contain all 3 of the base node types, so we put these matchers into 3 addMatcher calls in respect of which matcher returns which node type. The matchers `ExplicitCast`, `New`, `Delete`, `Argument`, `Unary`, `Dereference`, `BinaryLHS`, `BinaryRHS`, `If`, `While`, `For`, `Switch`, and `Return` should return statements, `StaticAssert`, `VarDecl`, and `CtorInits` should return declarations, and `Decltype`, `TemplateArg`, and `VLA` should return types. Now all we have to do is match for a simple call.

```
1   Finder->addMatcher(
2     traverse(
3       TK_IgnoreUnlessSpelledInSource,
4       stmt(eachOf(ExplicitCast, New, Delete, Argument, Unary,
            Dereference,
5               BinaryLHS, BinaryRHS, If, While, For, Switch, Return)))
6       .bind(Consume),
7     this);
8   Finder->addMatcher(traverse(TK_IgnoreUnlessSpelledInSource,
9                 decl(eachOf(StaticAssert, VarDecl, CtorInits)))
10                .bind(Consume),
11              this);
12  Finder->addMatcher(traverse(TK_IgnoreUnlessSpelledInSource,
13                type(eachOf(Decltype, TemplateArg, VLA)))
```

```
14                .bind(Consume),
15              this);
16
17  Finder->addMatcher(traverse(TK_IgnoreUnlessSpelledInSource, Call)
        , this);
```

Code 3.15: Separating the matchers to statement, declaration, and type.

The checker will match for all function calls (that are not declared `void` et cetera) bound to "call", and for the ones that are checked it will match once more with the bind "consume". For every match the callback happens and since matching "consume" deterministically comes before matching "call", we can be sure that if we matched with "consume" it is a checked call, and if we did not, it is an unchecked call so we can count the checks of our matched call expressions accordingly. In listing 3.16 we try to get our "consume" bound node for either of the 3 base node types, and if either gets a node back, we count it as a checked return. Otherwise we count it as unchecked.

```
1  void DiscardedReturnValueCheck::matchResult(
2    const MatchFinder::MatchResult &Result, bool ShouldCount) {
3    const auto *CE = Result.Nodes.getNodeAs<CallExpr>(Call);
4    assert(CE && "Bad matcher");
5
6    const void *ConsumeNode = nullptr;
7    if (const auto *D = Result.Nodes.getNodeAs<Decl>(Consume))
8      ConsumeNode = D;
9    if (const auto *S = Result.Nodes.getNodeAs<Stmt>(Consume))
10     ConsumeNode = S;
11   if (const auto *T = Result.Nodes.getNodeAs<Type>(Consume))
12     ConsumeNode = T;
13   if (ConsumeNode)
14     return registerCall(CE, CE->getDirectCallee(), ShouldCount,
         ConsumeNode);
15   if (ConsumedCalls.find(CE) == ConsumedCalls.end())
16   return registerCall(CE, CE->getDirectCallee(), ShouldCount,
       nullptr);
17 }
```

Code 3.16: Catching nodes of used return values.

Then the call expression is emplaced in our data structure for counting and because we do not want to count the same call twice, and if the call belongs to a function declaration that have previously been saved in our data structure, then we also increment its data in respect of it being checked or not.

At the end of the translation unit, we give our diagnosis.

3.2.4 Multipass phases

It is very clear that a checker like this is almost completely useless for projects larger than a single translation unit, unless we use the enhanced infrastructure, this makes the checker a perfect fit to test it. First we have to implement the new `virtual` functions.

`Collect` is very simple, it does the same thing as the old `check` callback did. Because of this, the logic of this step was put into a memeber function and is called in both functions, `check` and `collect`.

We save declarations and increment their data with each new call of the same function. At the end of a translation unit, `postCollect` writes this data into a new YAML file under the unique name that was generated by the infrastructure.

`Compact` reads in all the data from the collection files, fills up the data structure and then it writes the compacted data into the output YAML. Because both writing and reading into our data structure is necessary more than once in the checker, this also have been refactored, and `compact`, `postCollect` and `check` all calls these functions.

In listing 3.17 we can see, that we first extract the data from the compaction file into an optional `vector` of the function representation, and if the extraction actually happened, we emplace and modify the data of the files into the current state of the data structure. At the end, it returns whether succesful extraction happened or not. In listing 3.18 we see that `writeYAML` simply put our data into the output `stream`. These are both examples of possible writing and loading methods for YAML serialization that are easy to use and implement using LLVM's YAML library.

```
1 using FunctionVec = std::vector<SerializedFunction>;
2
3 static Optional<FunctionVec> loadYAML(StringRef File) {
4   using namespace llvm;
5
```

```
  ErrorOr<std::unique_ptr<MemoryBuffer>> IStream =
    MemoryBuffer::getFileAsStream(File);
  if (!IStream)
    return None;

  FunctionVec R;
  yaml::Input YIn{**IStream};
  YIn >> R;

  return R;
}

static bool loadYAML(StringRef FromFile,
             DiscardedReturnValueCheck::FunctionMapTy &ToMap) {
  Optional<FunctionVec> OV = loadYAML(FromFile);
  if (!OV)
    return false;

  for (const SerializedFunction &SF : *OV) {
    DiscardedReturnValueCheck::Function &F =
      ToMap
        .try_emplace(SF.ID,
                DiscardedReturnValueCheck::Function{0, 0, nullptr,
                    {}})
        .first->second;

    F.ConsumedCalls += SF.ConsumedCalls;
    F.TotalCalls += SF.TotalCalls;
  }

  return true;
}
```

Code 3.17: Functions for loading.

```
static void writeYAML(StringRef Whence, FunctionVec Elements,
                      StringRef ToFile) {
  std::error_code EC;
  llvm::raw_fd_ostream FS(ToFile, EC, llvm::sys::fs::OF_Text);
  if (EC) {
    llvm::errs() << "DiscardedReturnValueCheck: Failed to write "
```

```
        << Whence
7                   << " output file: " << EC.message();
8     llvm::report_fatal_error("", false);
9     return;
10  }
11
12  llvm::yaml::Output YAMLOut(FS);
13  YAMLOut << Elements;
14 }
```

Code 3.18: Function for writing.

Both `postCollect` and `compact` uses `writeYAML` with the YAML compatible representation of the function representation. For the purpose of writing, these functions both use a refactored function that converts the latter into the YAML compatible version. These are seen in listing 3.19 with the method of making a `struct` YAML compatible as well.

```
1  struct SerializedFunction {
2    std::string ID;
3    std::size_t ConsumedCalls, TotalCalls;
4  };
5
6  template <> struct MappingTraits<SerializedFunction> {
7    static void mapping(IO &IO, SerializedFunction &F) {
8      IO.mapRequired("ID", F.ID);
9      IO.mapRequired("Consumed", F.ConsumedCalls);
10     IO.mapRequired("Total", F.TotalCalls);
11   }
12 };
13
14 LLVM_YAML_IS_SEQUENCE_VECTOR(SerializedFunction)
15
16 static FunctionVec
17 yamlize(const DiscardedReturnValueCheck::FunctionMapTy &Map) {
18   FunctionVec SFs;
19   llvm::transform(Map, std::back_inserter(SFs), [](auto &&E) {
20     return SerializedFunction{E.first().str(), E.second.
         ConsumedCalls,
21                               E.second.TotalCalls};
22   });
```

```
23   return SFs;
24 }
```

Code 3.19: Steps for making the data YAML writable.

Check works almost exactly the same way. First we check if compact has happened already. If not, we give diagnosis per translation units, the same way as before. If collection and compacting has happened for this cheker we fill up our data structure, but only if this step has not happened before. For this, we use the same refactored functions as in `collect`, `matchResult` and thus `registerCalls`, with the exception, that incrementation for each call does not need to happen, since all the data on the amount of checks and calls have been acquired before, from the compacted file. Then we take the unique name (as a `string`) of a function declaration, or generate one if one does not already exist. After this we check again, whether or not we stored this declaration in the data structure, and if not, we store it, but only if its call was unused, and then if compact has not happened, we increment its data accordingly. If it has, we have nothing left to do. Now we only need to give diagnosis to the right calls, which happens at the end of each translation unit.

3.3 Evaluation

This checker's upgrade to the multipass infrastructure was done by a simple logic, and it differs from the original one only by a little. However it gives us very different results. The checker has run on large projects in different ways. It ran with 50% and 80% treshold both with and without collection and compacting. These were the results:

Project name	**Multiple phase used**	**Treshold**	**Amount of warnings**
Bitcoin `v0.20.1`[16]	✗	50%	140
	✓	50%	320
	✗	80%	25
	✓	80%	92

Project name	Multiple phase used	Treshold	Amount of warnings
CodeChecker `v6.17.0`[14]	✗	50%	2
	✓	50%	5
	✗	80%	0
	✓	80%	0
Contour `v0.2.0.173`[17]	✗	50%	38
	✓	50%	107
	✗	80%	10
	✓	80%	8
cUrl `7.66.0`[18]	✗	50%	55
	✓	50%	210
	✗	80%	6
	✓	80%	104
FFmpeg `n4.3.1`[19]	✗	50%	904
	✓	50%	1846
	✗	80%	148
	✓	80%	613
WebM `1.0.0.27`[20]	✗	50%	9
	✓	50%	10
	✗	80%	0
	✓	80%	1
LLVM-Project `12.0.0`[21]	✗	50%	4615
	✓	50%	6837
	✗	80%	1077
	✓	80%	2150
Memcached `1.6.8`[22]	✗	50%	32
	✓	50%	49
	✗	80%	5
	✓	80%	16
MongoDB `r4.4.6`[23]	✗	50%	1671
	✓	50%	2463
	✗	80%	370
	✓	80%	1223

Project name	Multiple phase used	Treshold	Amount of warnings
OpenSSL `3.0.0`[24]	✗	50%	427
	✓	50%	1060
	✗	80%	56
	✓	80%	244
PostgreSQL `r13.0`[25]	✗	50%	799
	✓	50%	644
	✗	80%	62
	✓	80%	215
Protocol Buffers `v3.13.0`[26]	✗	50%	51
	✓	50%	122
	✗	80%	20
	✓	80%	75
Qt Base `v6.2.0`[27]	✗	50%	1195
	✓	50%	2764
	✗	80%	200
	✓	80%	1093
SQLite `3.33.0`[28]	✗	50%	284
	✓	50%	288
	✗	80%	42
	✓	80%	48
tmux `2.6`[29]	✗	50%	35
	✓	50%	48
	✗	80%	1
	✓	80%	1
twin `v0.8.1`[30]	✗	50%	54
	✓	50%	69
	✗	80%	9
	✓	80%	22
Vim `v8.2.1920`[31]	✗	50%	227
	✓	50%	383
	✗	80%	32
	✓	80%	54

Project name	Multiple phase used	Treshold	Amount of warnings
Xerces `v3.2.3`[32]	×	50%	125
	✓	50%	197
	×	80%	9
	✓	80%	38

Table 3.1: Results of different runs on live projects.

In section 3.3 we can see that the numbers between both single and multiple phase, and 50% and 80% treshold runs largely differ in a considerable amount of test projects. There were results, where the multiple phase run gave more warnings, indicating the aforementioned bugs hiding in separate translation units only, and some results where it gave less warnings, indicating false positives in single runs.

In fig. 3.6 and fig. 3.7 we can see some results in bigger detail. Each coloumn represents a function, with its height as the total amount of calls. The green and red parts represent the checked and unchecked calls. The descending graph of black dots display the ratio of checked to all calls, which is important regarding our treshold.

In these runs, the amount of separate unchecked functions were similar, but in the multi run, the amount of calls for these functions was obviously a lot higher. In the single run, the ratios are closer to the 80% treshold, and most functions here were called less than 20 times, and were only unchecked once or twice. However if we look at the multi run, the ratios are a lot higher. They were unchecked only a couple of times in this run as well, but the amount of total calls per function was bigger than in the previous run, resulting in the functions staying more consistently above our 80% treshold.

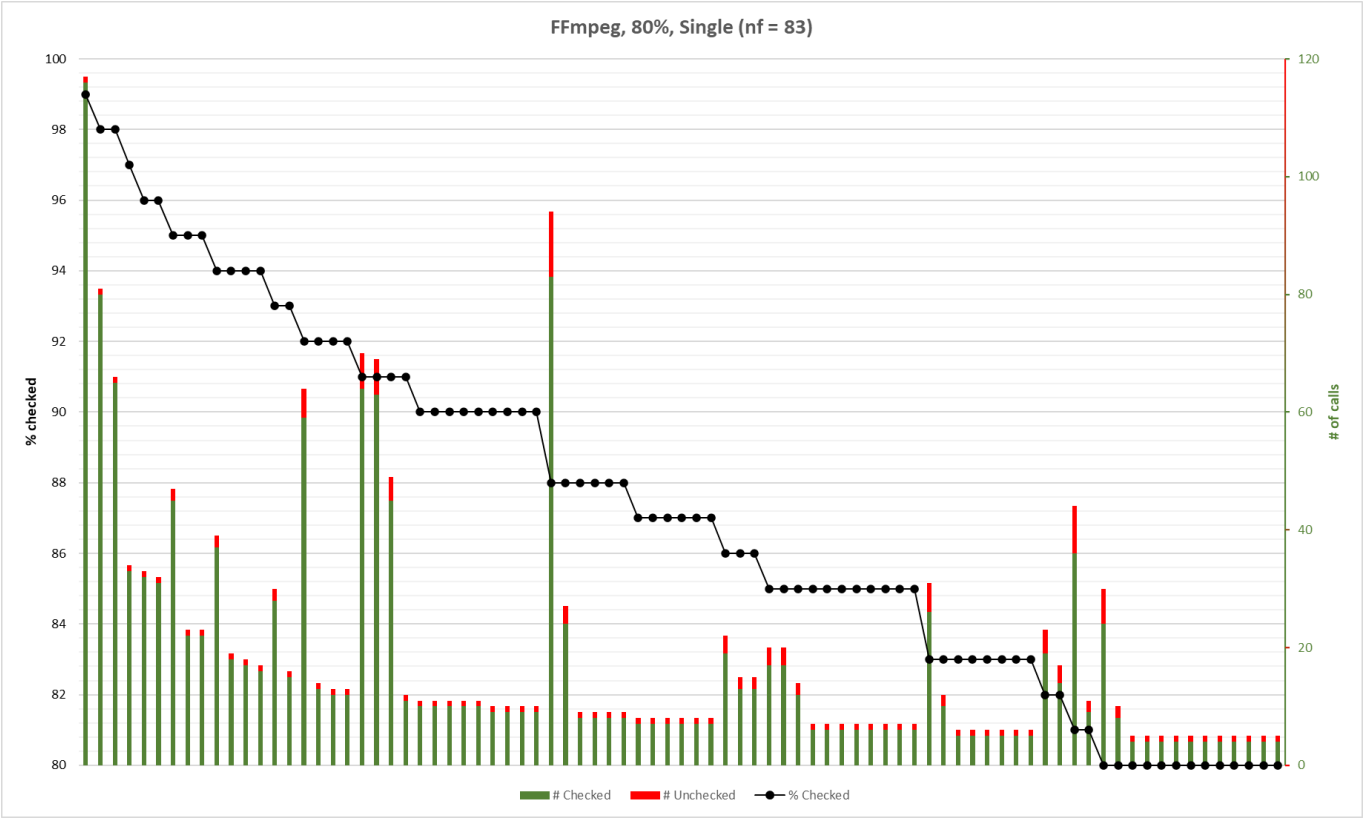


Figure 3.6: Diagram of 80% treshold single run on FFmpeg.

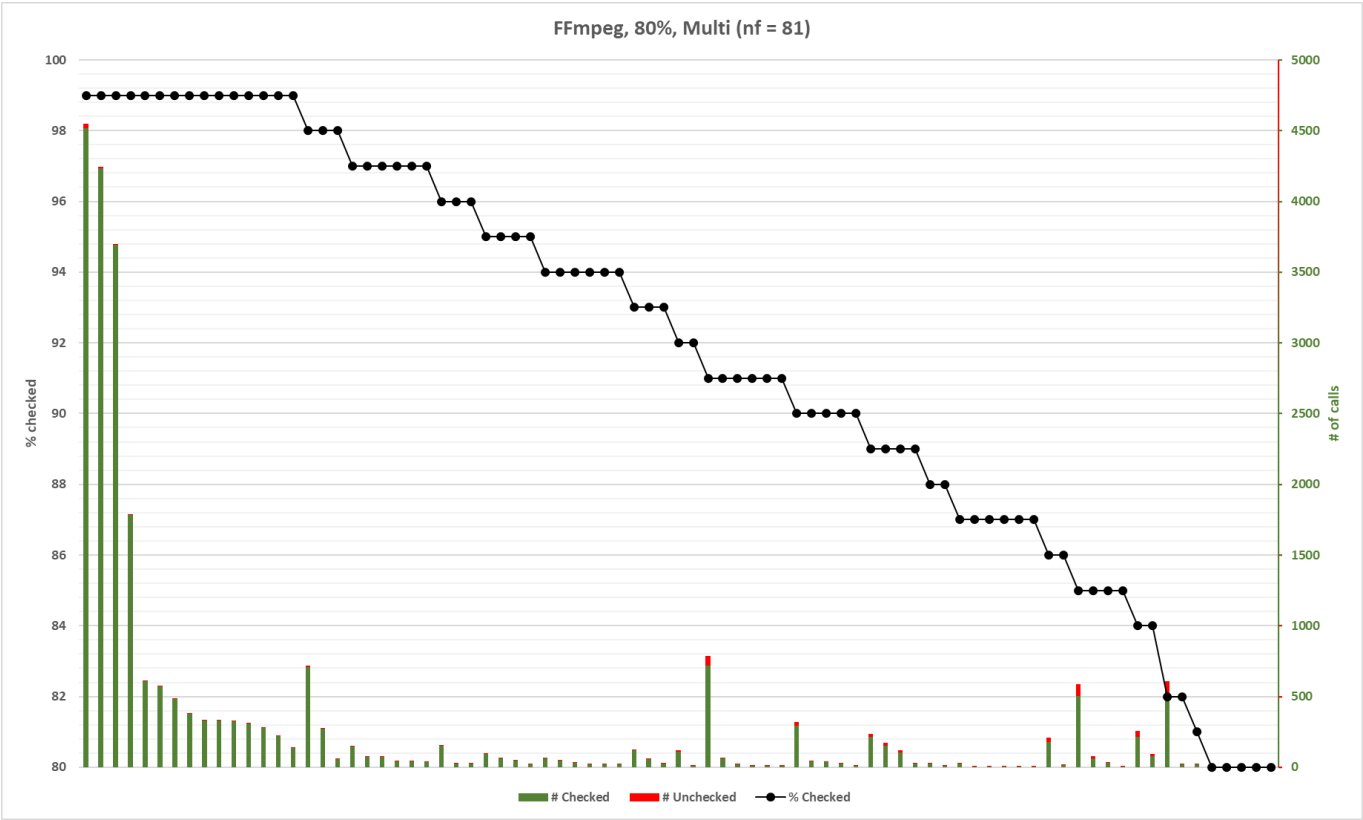


Figure 3.7: Diagram of 80% treshold multi run on FFmpeg.

Let us take a look at a 50% treshold run comparison as well. In fig. 3.8 and fig. 3.9 we get very different results again. Here we can still see the higher ratios in the multi run, as well as higher overall amount of function calls. This time we have more unchecked calls for functions that have been called considerably more than the rest, in both amount and ratio.

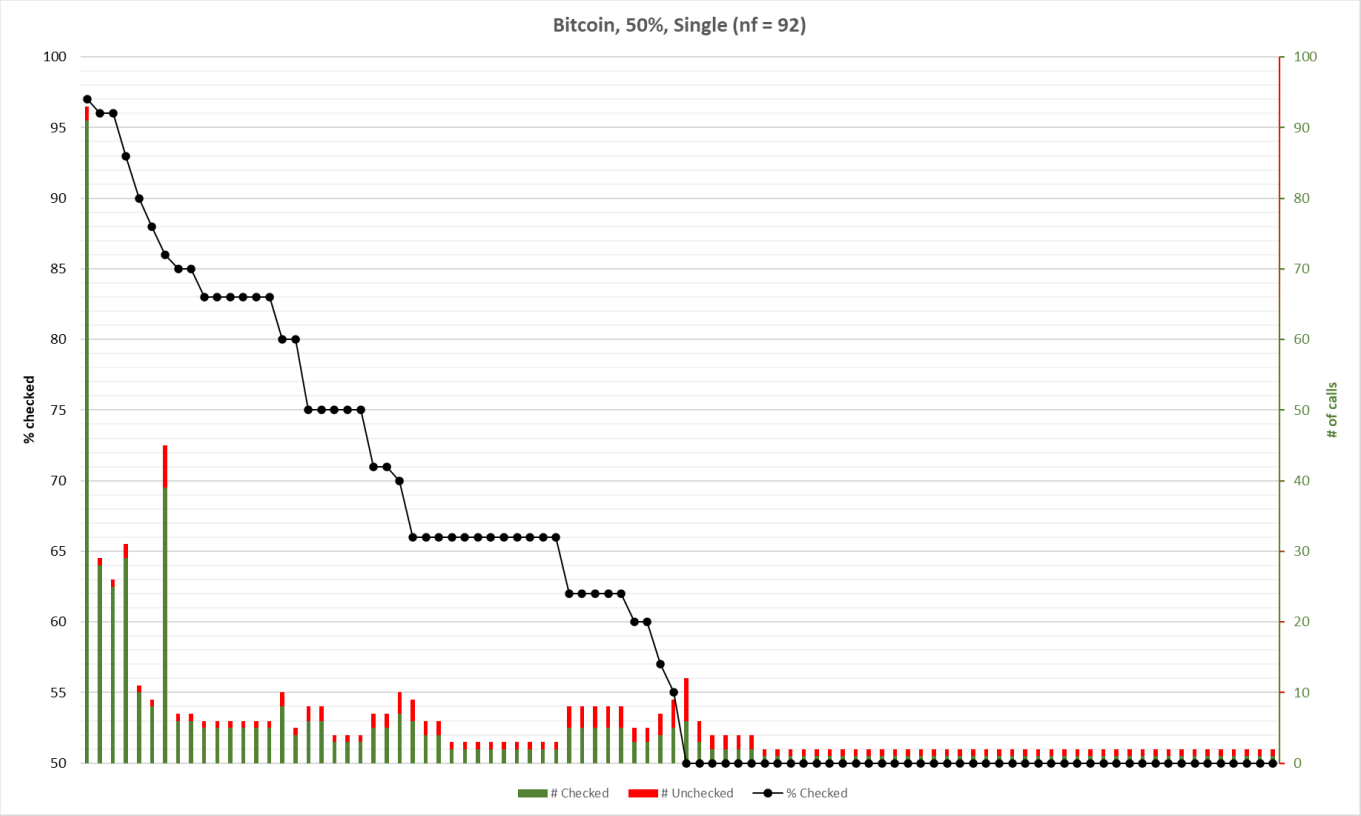


Figure 3.8: Diagram of 50% treshold single run on Bitcoin.

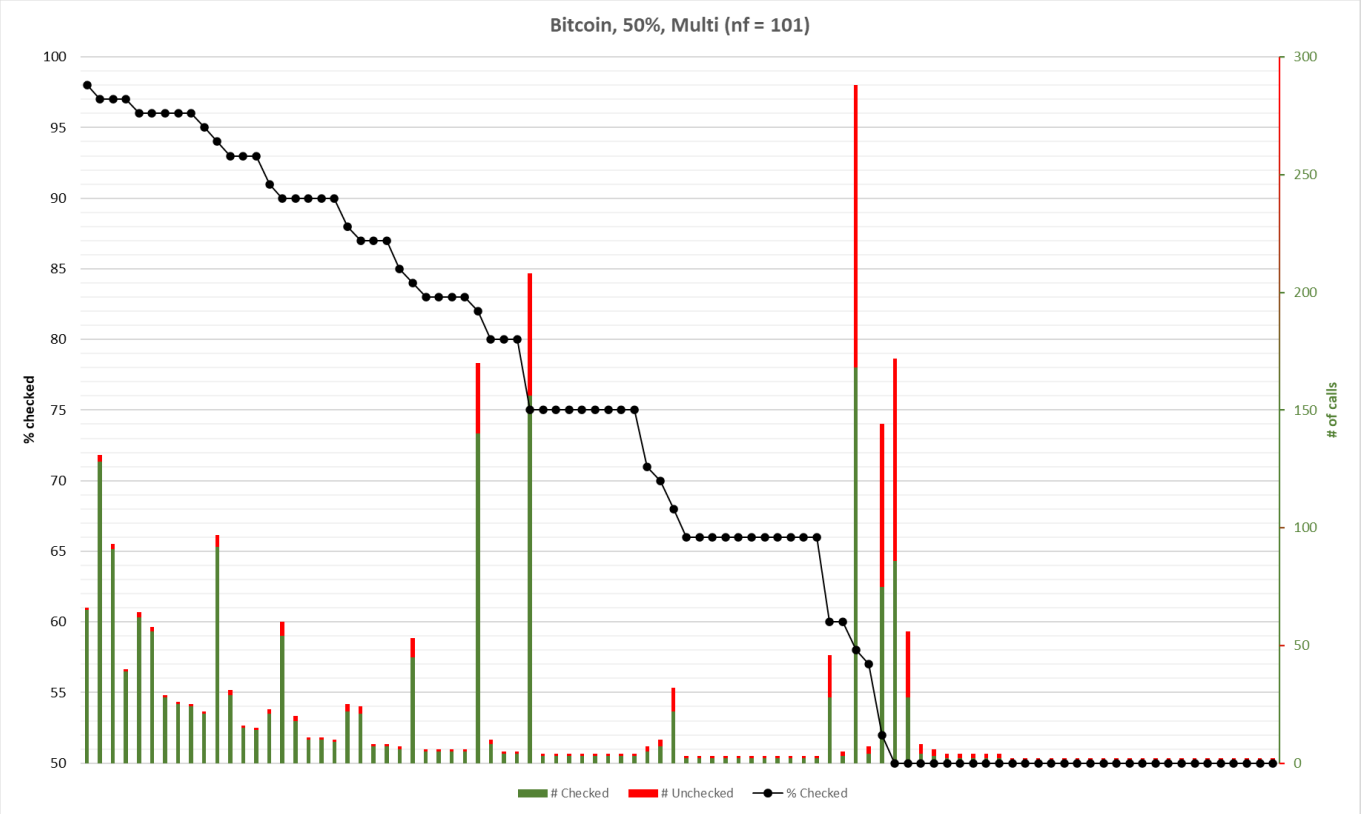


Figure 3.9: Diagram of 50% treshold multi run on Bitcoin.

The diagrams for all of the aforementioned project tests can be found in appendix A. What can be said about almost all of the larger results is that they have a great amount of functions barely passing the treshold whose total amount of calls do not exceed a considerable number compared to the size of the project and some other functions within the results. These could potentially be ignored in some cases.

Chapter 4

Conclusion

This thesis demonstrated the problem of unchecked return values, to which problem it gave a solution in form of a statistical method without actually modifying the source code of said unchecked return values. This solution was made as an implementation in the Clang-Tidy pattern matching based static analysis library, however the statistical method would fail because of the way Clang-Tidy checkers work, separately for each translation unit. This leads to false statistics. In order to fix this, I have enhanched the infrastructure and created the option of multiphase analysis, that will collect the needed information per translation unit, accumulate these and make a project level set of information, and finally give diagnosis with this newly compacted information. The finished checker was accomplished using this new infrastructure. This checker method can obviously run without the use of multiphase analysis, because the new infrastructure is made to be backwards compatible with the single phase mode, but the new infrastructure was vital for the statistics of the checker. This proved the checker to be an excellent example for the usage and utility of the multiphase mode.

4.1 Future Works

Previously in section 3.3 I talked about ignoring a set of functions whose total amount of calls do not exceed a given value. This could potentially be implemented into the checker as an option, so that the user could give the checker the number, and the checker will not give warnings to a function call if the function's amount of calls is less than said number. It would make the diagnosis clearer in some cases.

The first possible future work ignores functions based on their statistics, but ignoring based on function property could be useful too. The checker should chategorically ignore some functions. There are also functions whose return value can not be always checked. A classic example would be the << operator or `std::append`, since the first will always return the `stream`, and the latter will always return `*this` for continuous use of the same function. `std::cout << '1 ' << '2 ' << '3' << std::endl ;` will always be unchecked in the end, since the last << can not be checked (with merit) without creating another unchecked return value.

The hash generation for the YAML filename does not respect multiple compilations of the same file with different compile flags. This is something that could be fixed in the future as well.

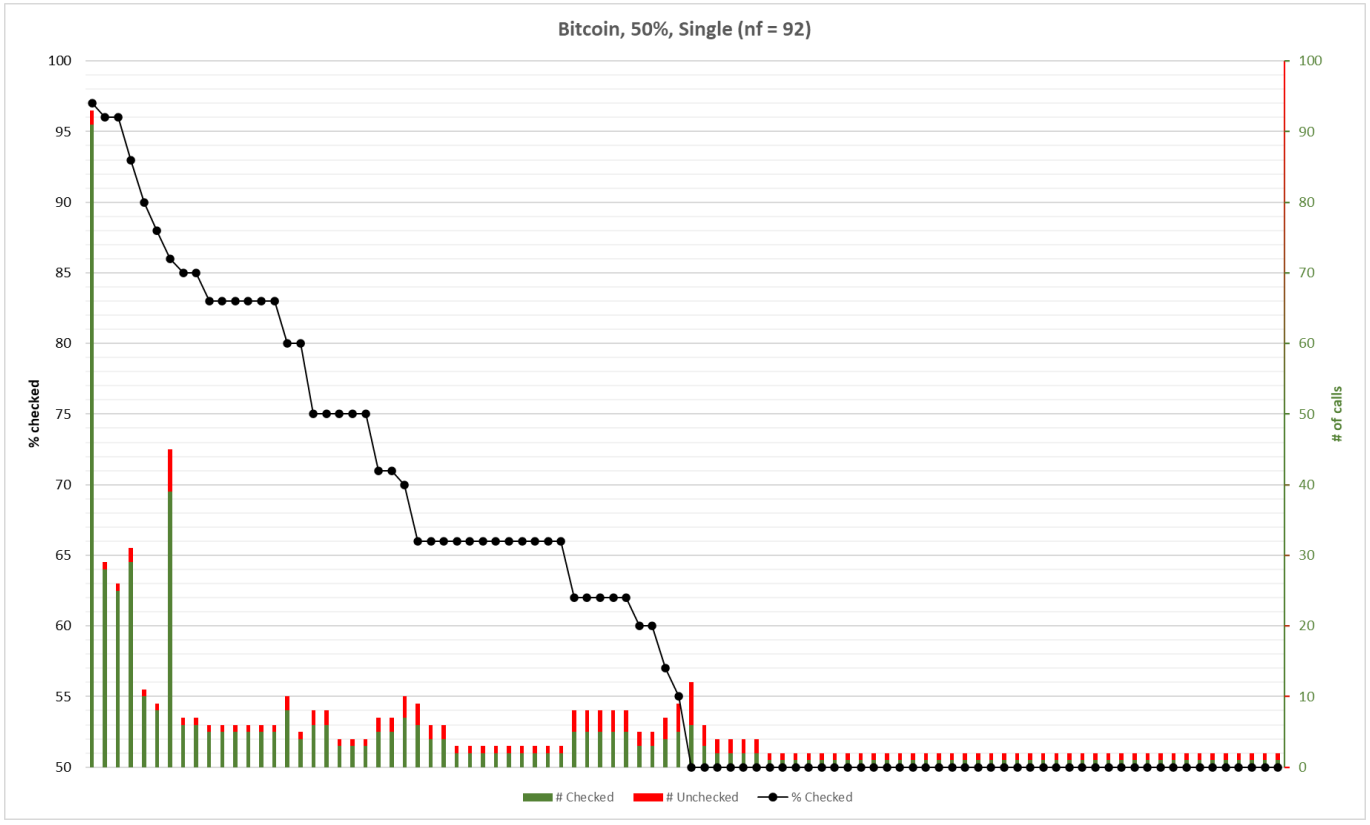
Acknowledgements

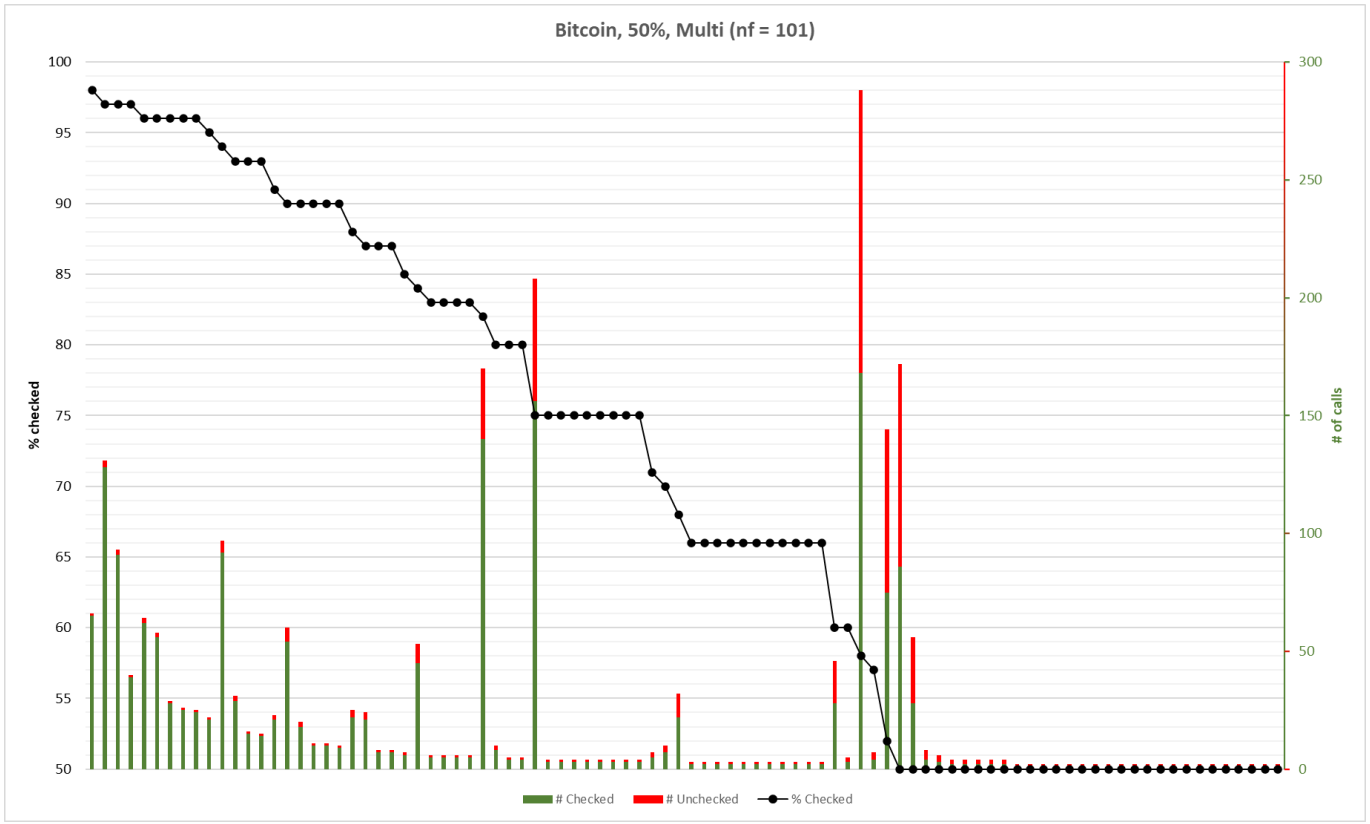
I would like to express my deepest appreciation to my supervisor, Richárd Szalay for his invaluable patience and feedback, and without whom I could not have completed this thesis in time. I am also greatful to my professor, Dr. Zoltán Porkoláb, who generously provided his knowledge, expertise and inspiration.

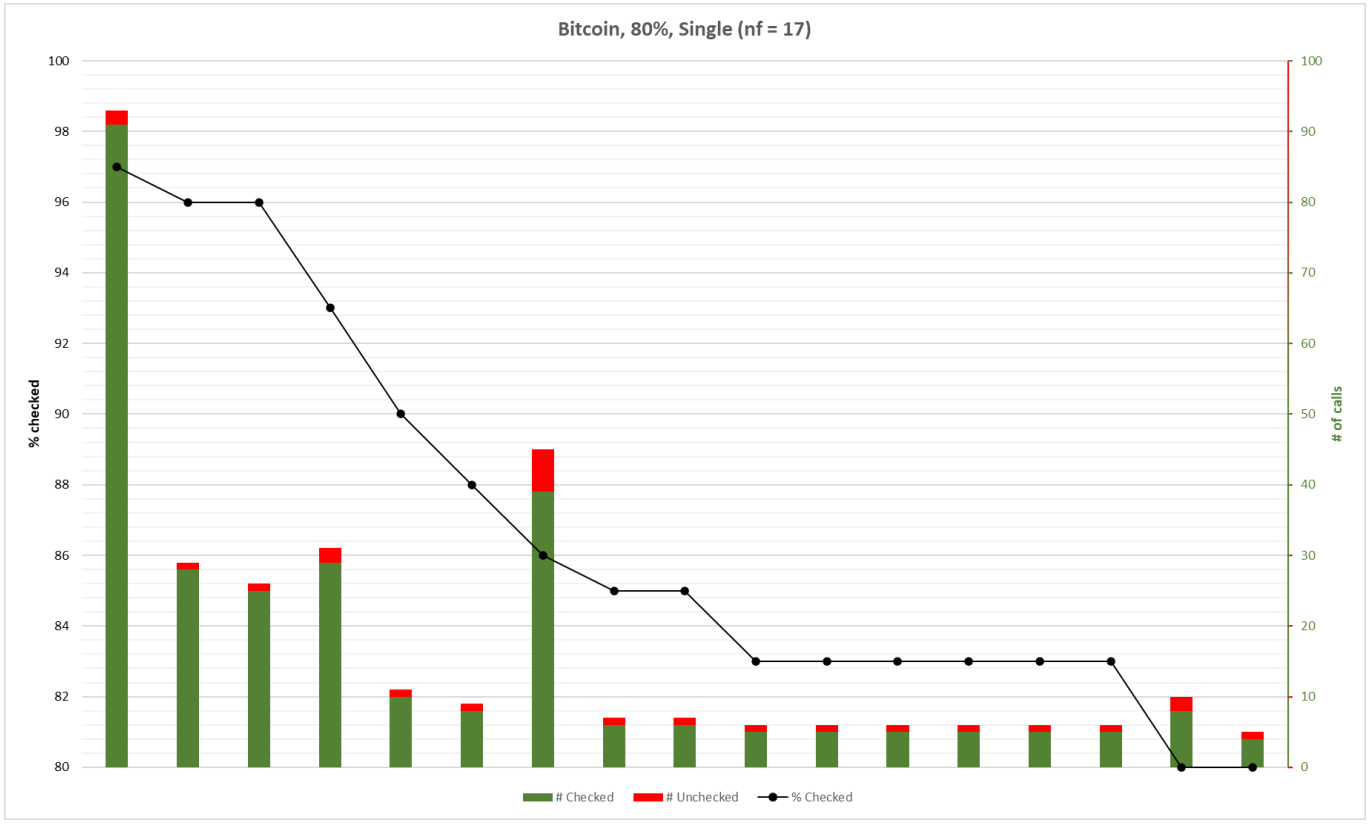
Appendix A

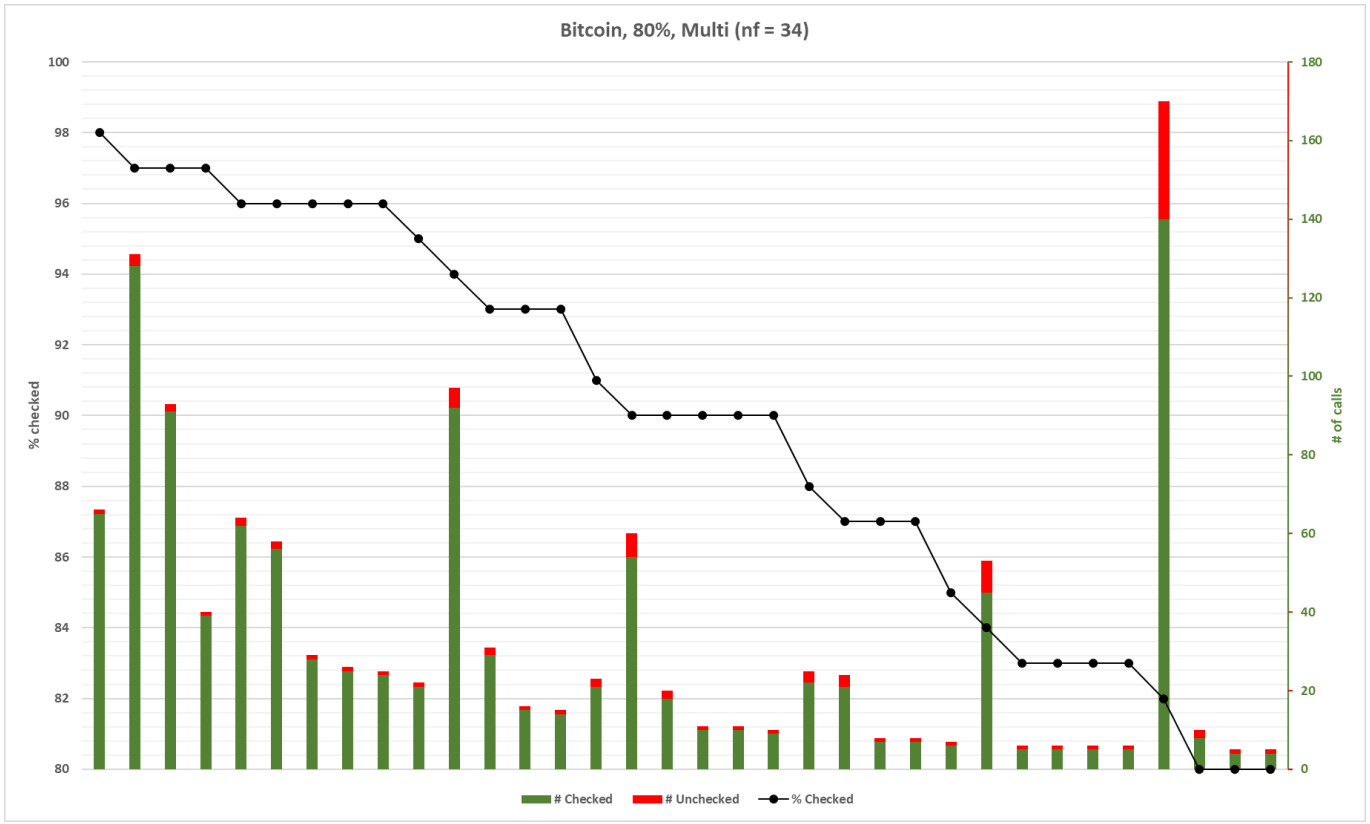
Project results

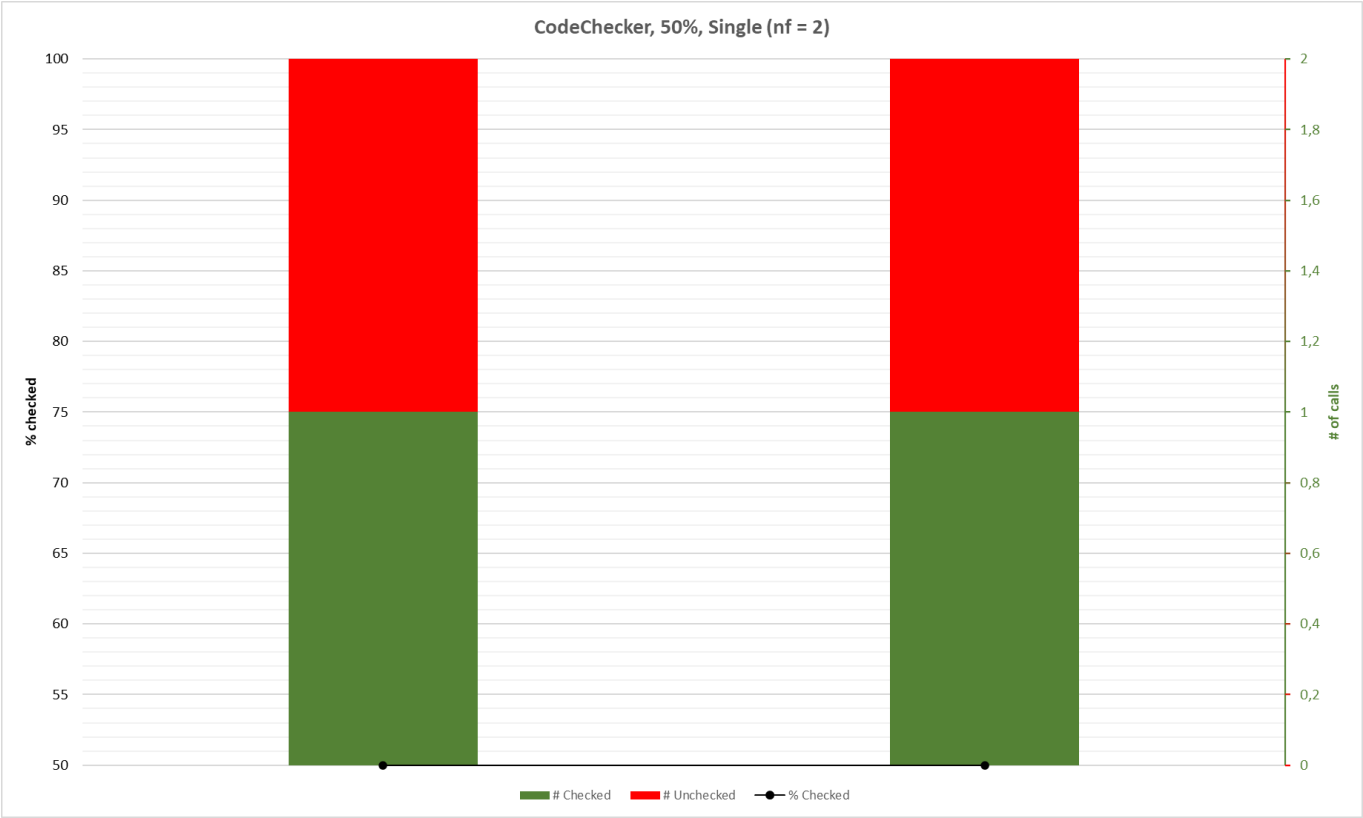
The Discarded Return Value Checker was run on 18 different live project with 4 different setting. Two using the new multiple phase infrastructure and two with the single phase one, on 50% treshold and on 80% treshold. These are the results of the runs regarding the amount of different functions found, their amount of calls both checked and unchecked, and the ratio of these two amounts.


Bitcoin, 50%, Single (nf = 92)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
100
90
80
70
60
50
40
30
20
10
0
Checked
Unchecked
% Checked


Bitcoin, 50%, Multi (nf = 101)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
300
250
200
150
100
50
0
Checked
Unchecked
% Checked


Bitcoin, 80%, Single (nf = 17)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
100
90
80
70
60
50
40
30
20
10
0
Checked
Unchecked
% Checked

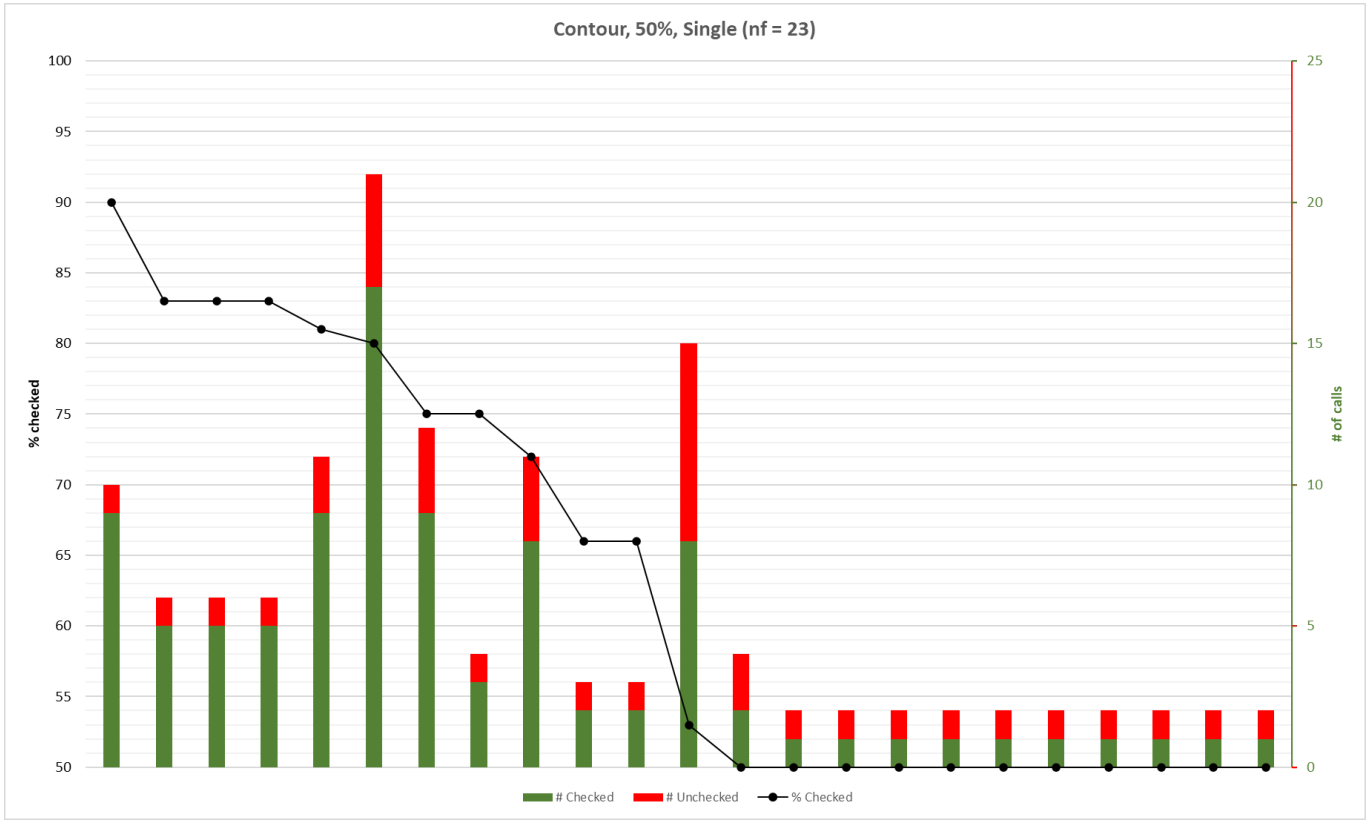

Bitcoin, 80%, Multi (nf = 34)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
180
160
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

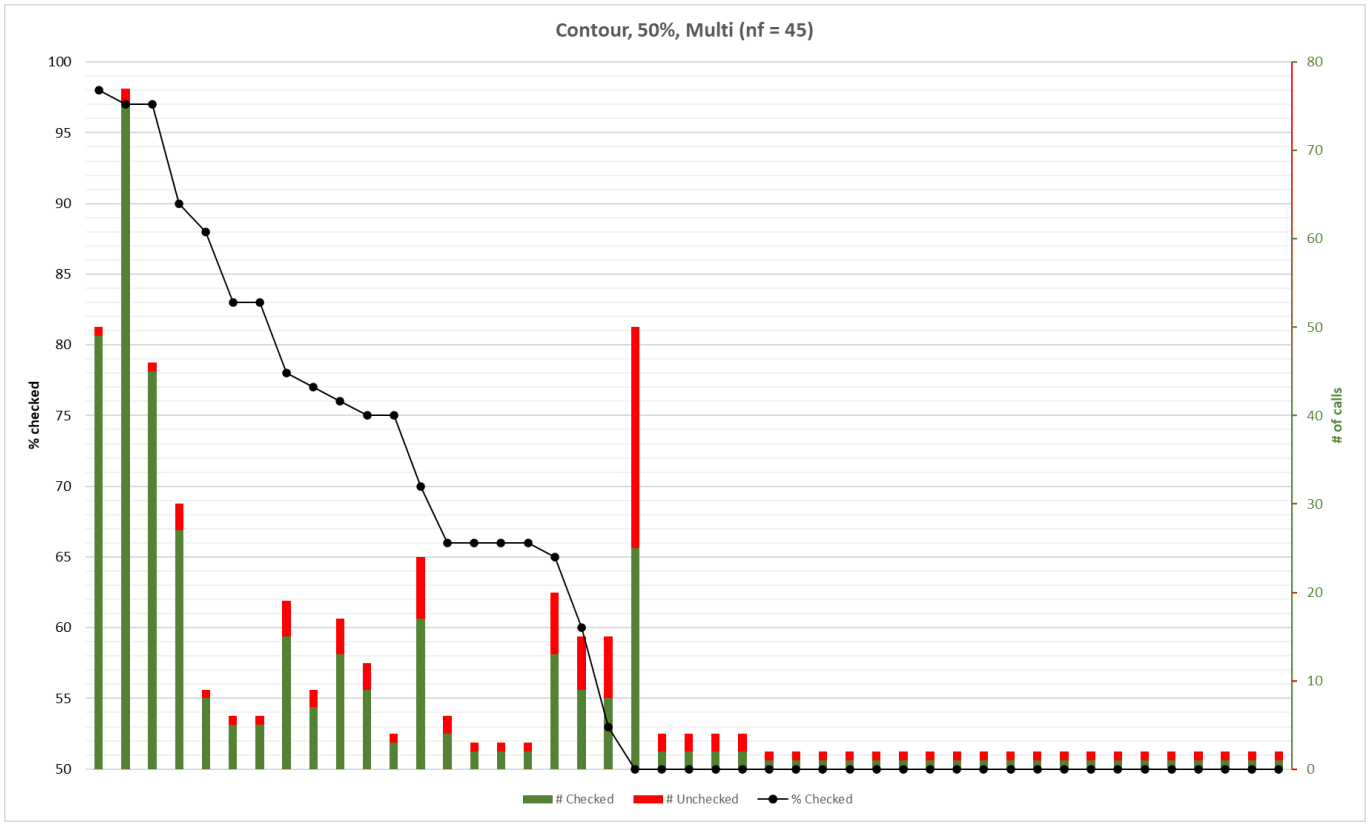

CodeChecker, 50%, Single (nf = 2)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
2
1,8
1,6
1,4
1,2
1
0,8
0,6
0,4
0,2
0
Checked
Unchecked
% Checked


CodeChecker, 50%, Multi (nf = 3)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
12
10
8
6
4
2
0
Checked
Unchecked
% Checked


CodeChecker, 80%, Single (nf = 0)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
1
0,8
0,6
0,4
0,2
0
Checked
Unchecked
% Checked

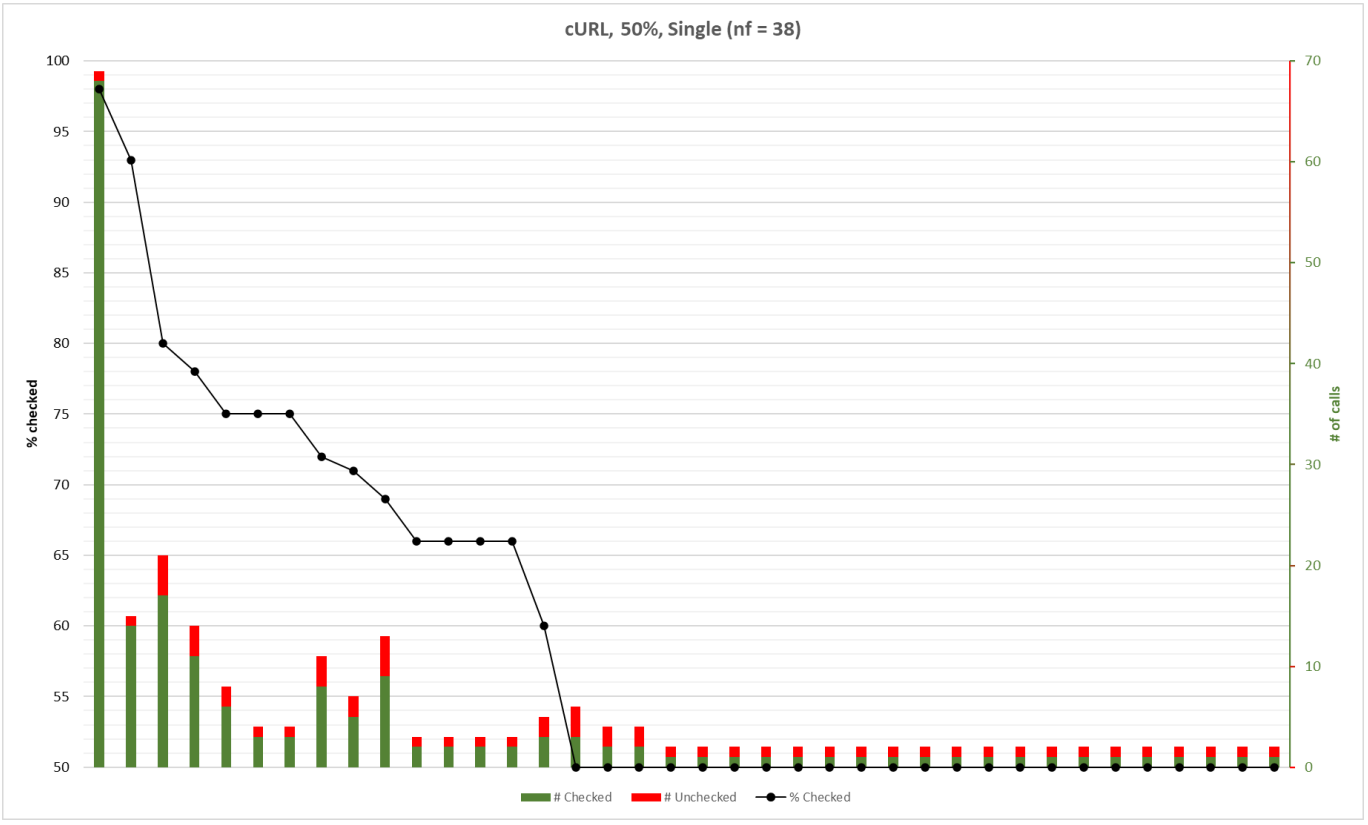

CodeChecker, 80%, Multi (nf = 0)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
1
0,8
0,6
0,4
0,2
0
Checked
Unchecked
% Checked


Contour, 50%, Single (nf = 23)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
25
20
15
10
5
0
Checked
Unchecked
% Checked

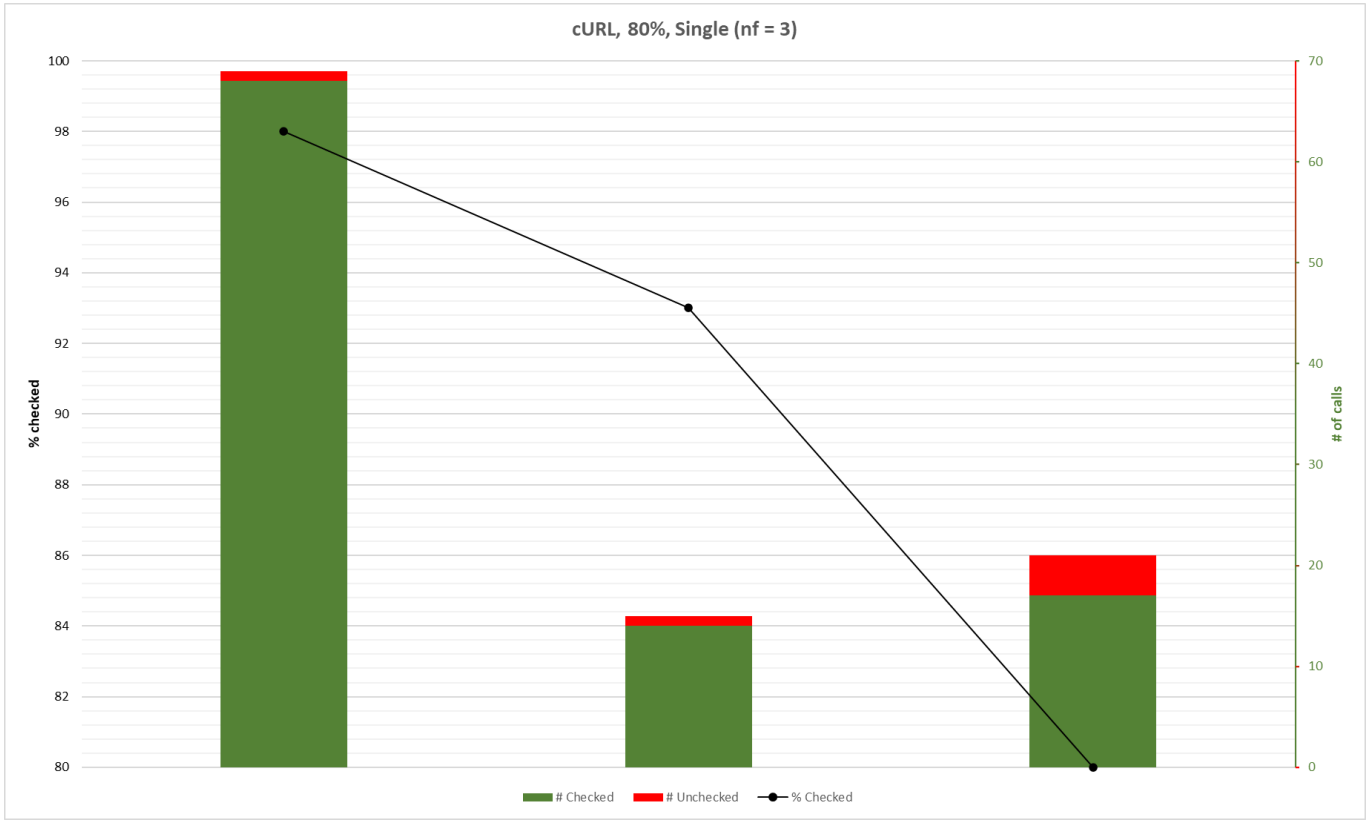

Contour, 50%, Multi (nf = 45)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
80
70
60
50
40
30
20
10
0
Checked
Unchecked
% Checked

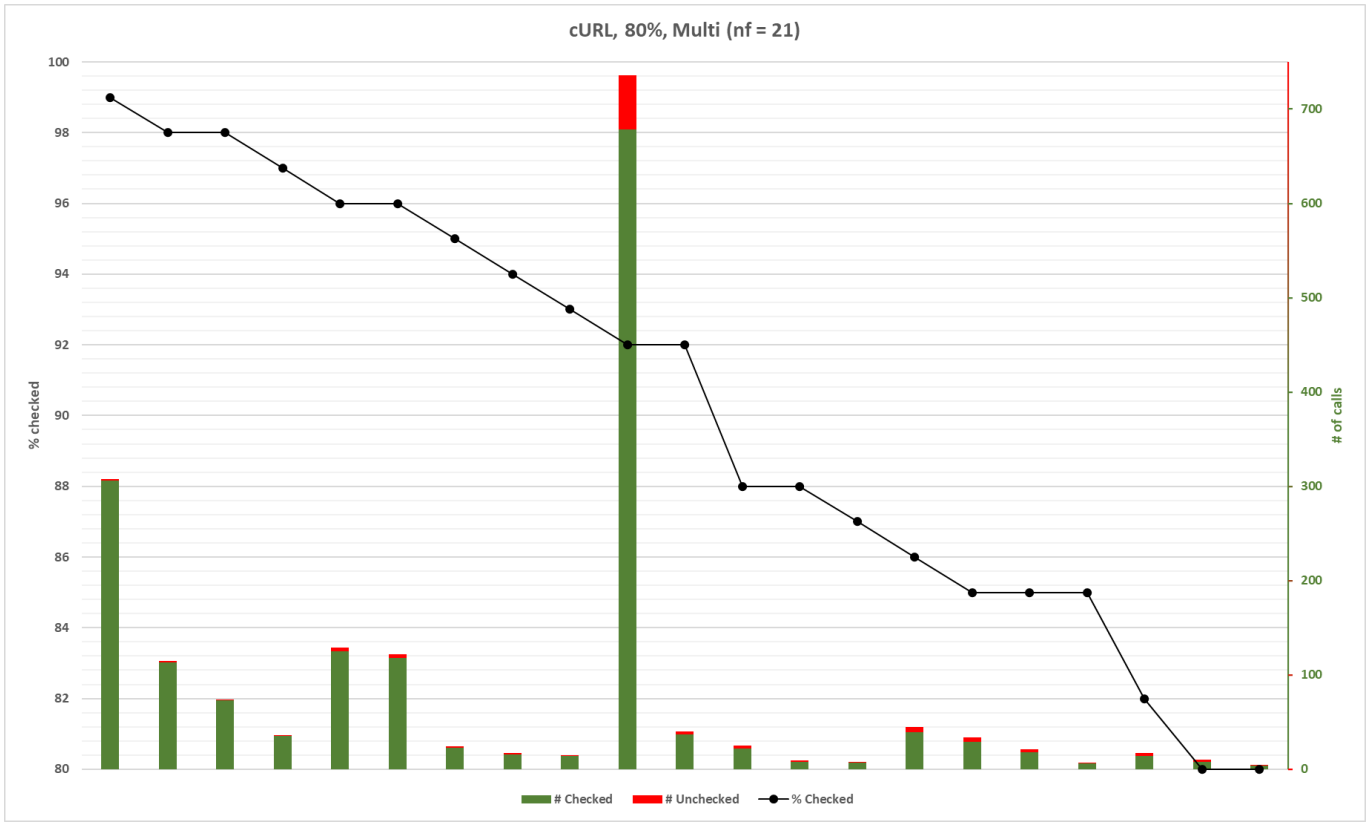

Contour, 80%, Single (nf = 6)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
25
20
15
10
5
0
Checked
Unchecked
% Checked

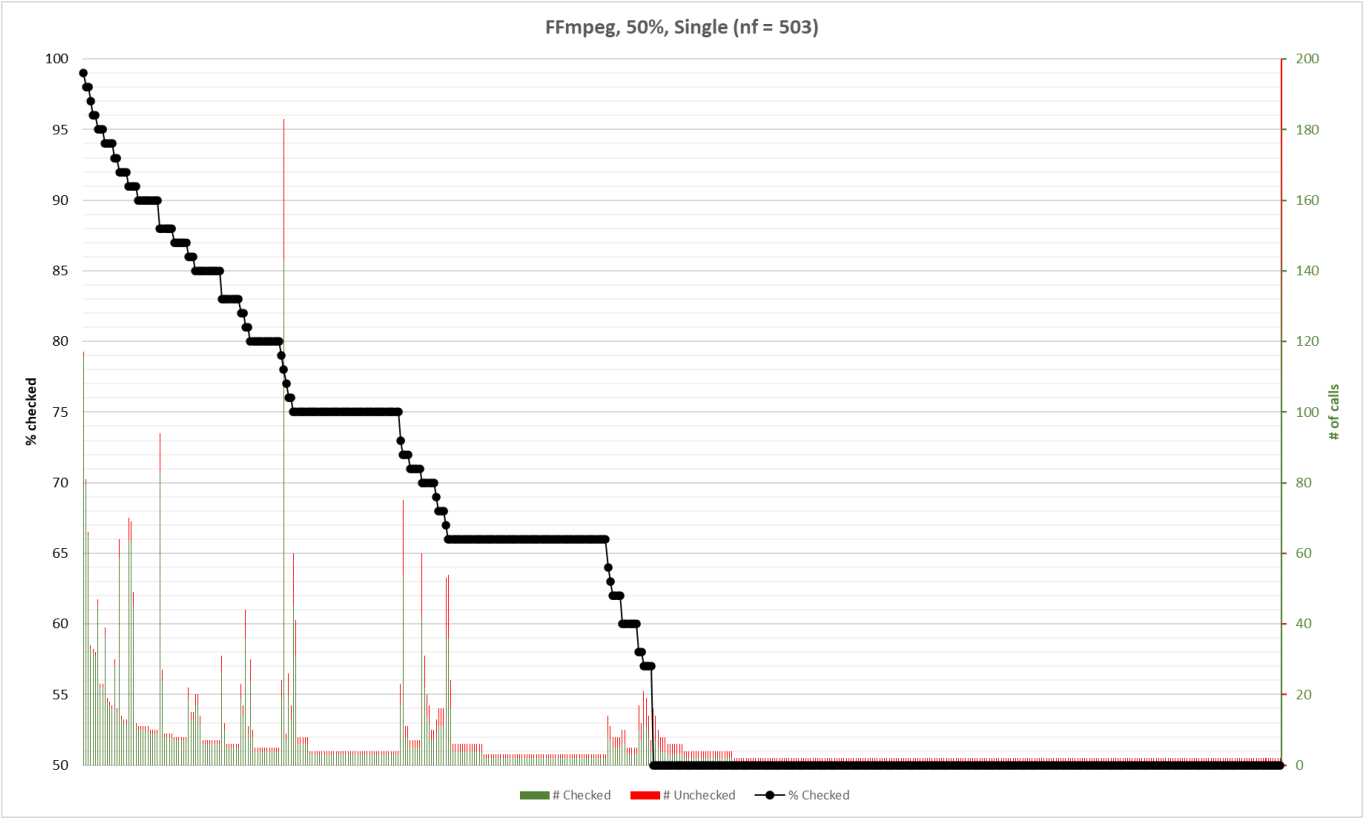

Contour, 80%, Multi (nf = 7)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
80
70
60
50
40
30
20
10
0
Checked
Unchecked
% Checked

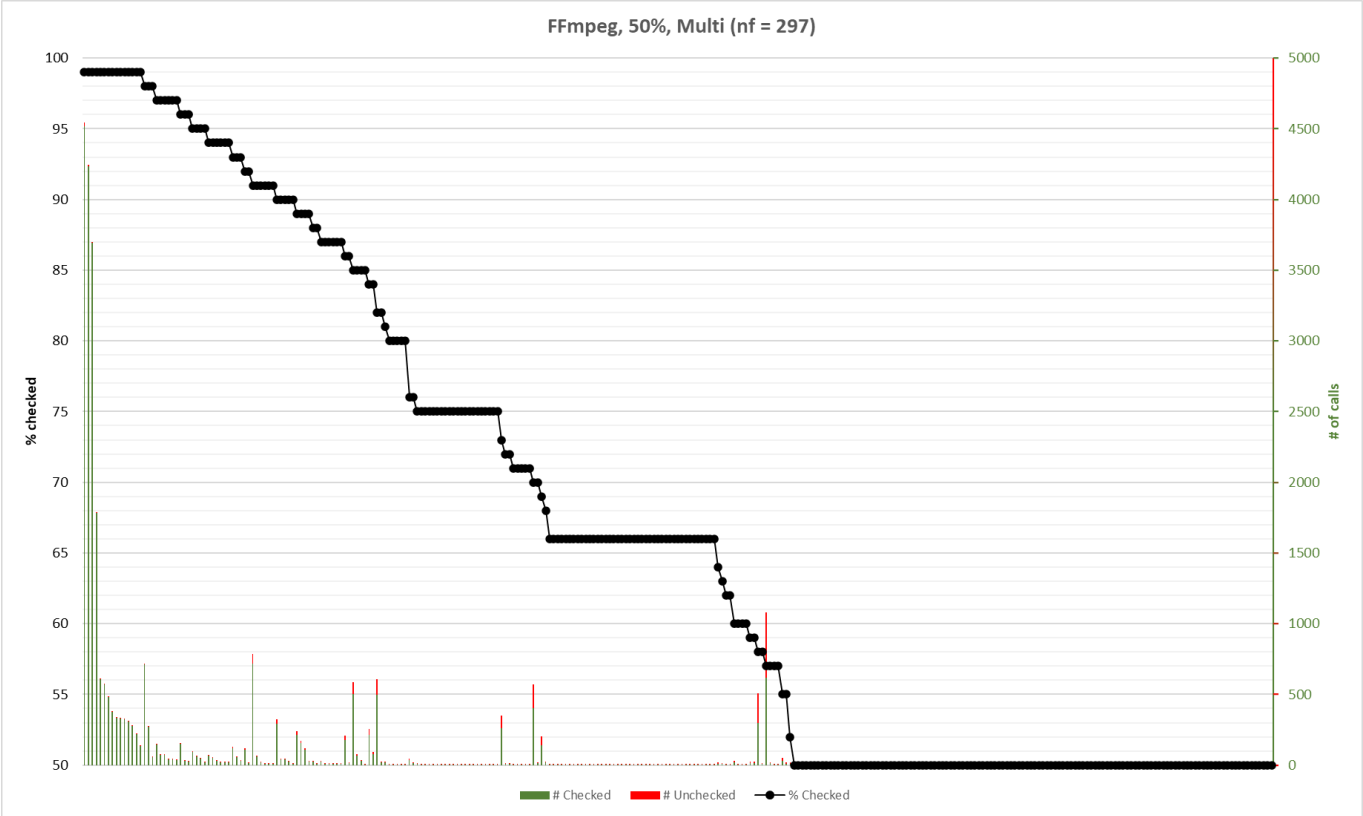

cURL, 50%, Single (nf = 38)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
70
60
50
40
30
20
10
0
Checked
Unchecked
% Checked

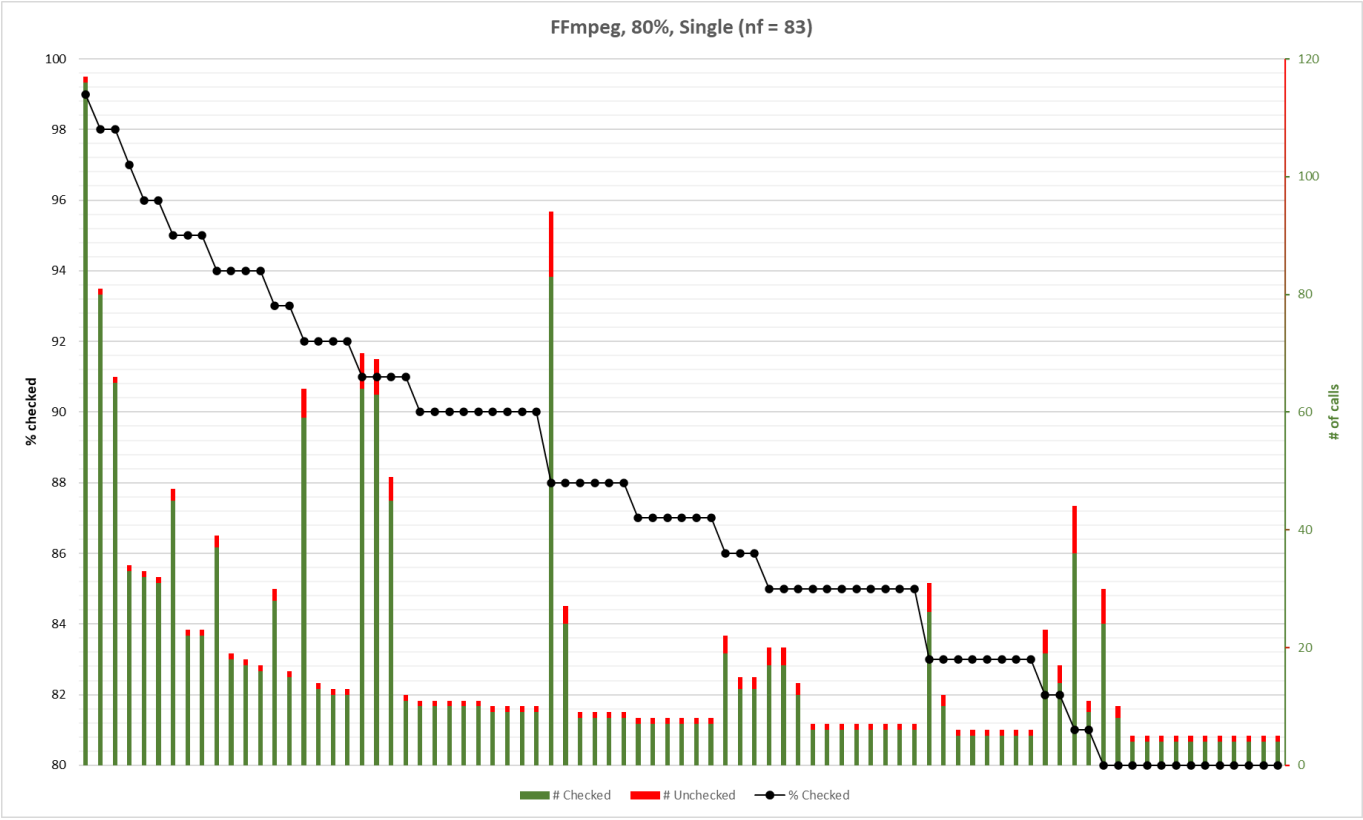

cURL, 50%, Multi (nf = 58)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
700
600
500
400
300
200
100
0
Checked
Unchecked
% Checked

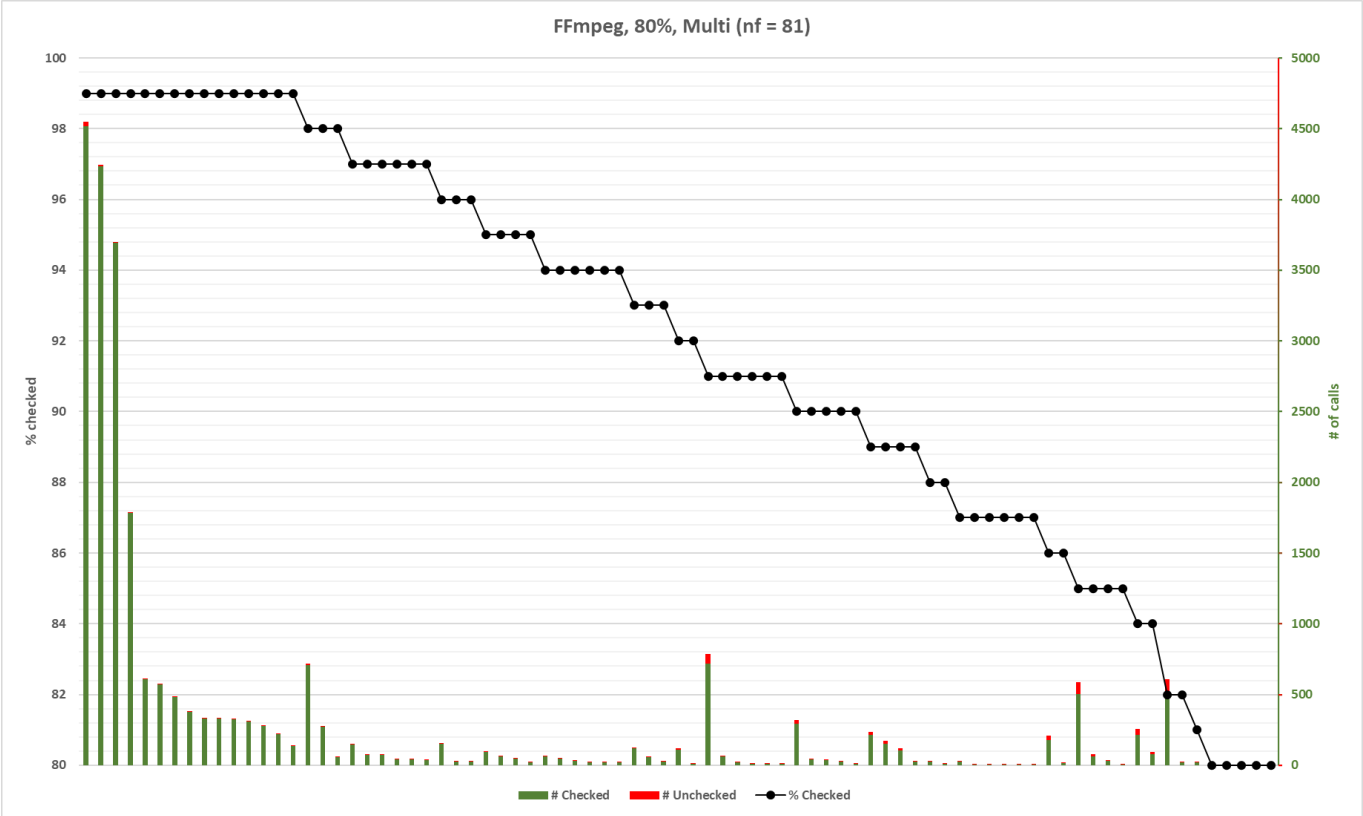

cURL, 80%, Single (nf = 3)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
70
60
50
40
30
20
10
0
Checked
Unchecked
% Checked

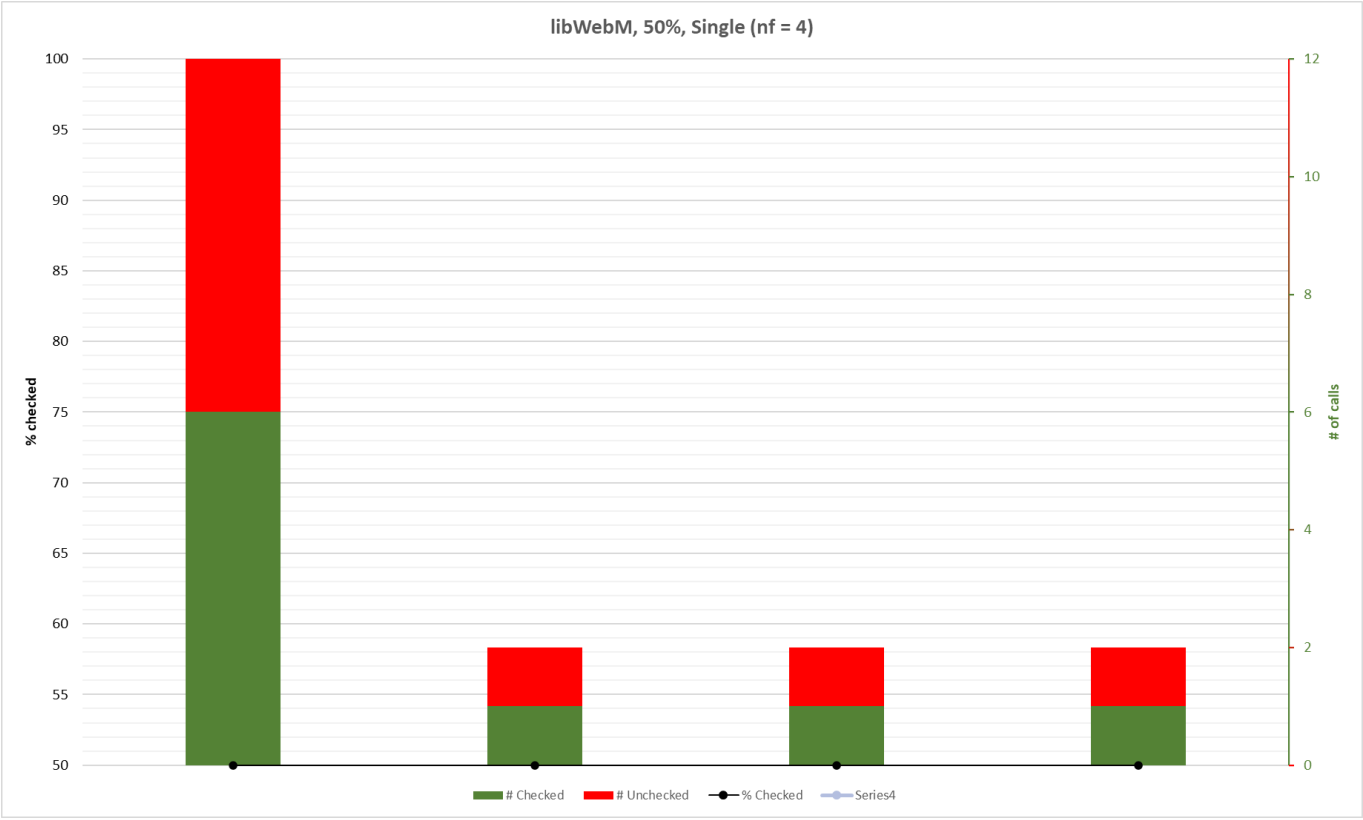

cURL, 80%, Multi (nf = 21)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
700
600
500
400
300
200
100
0
Checked
Unchecked
% Checked

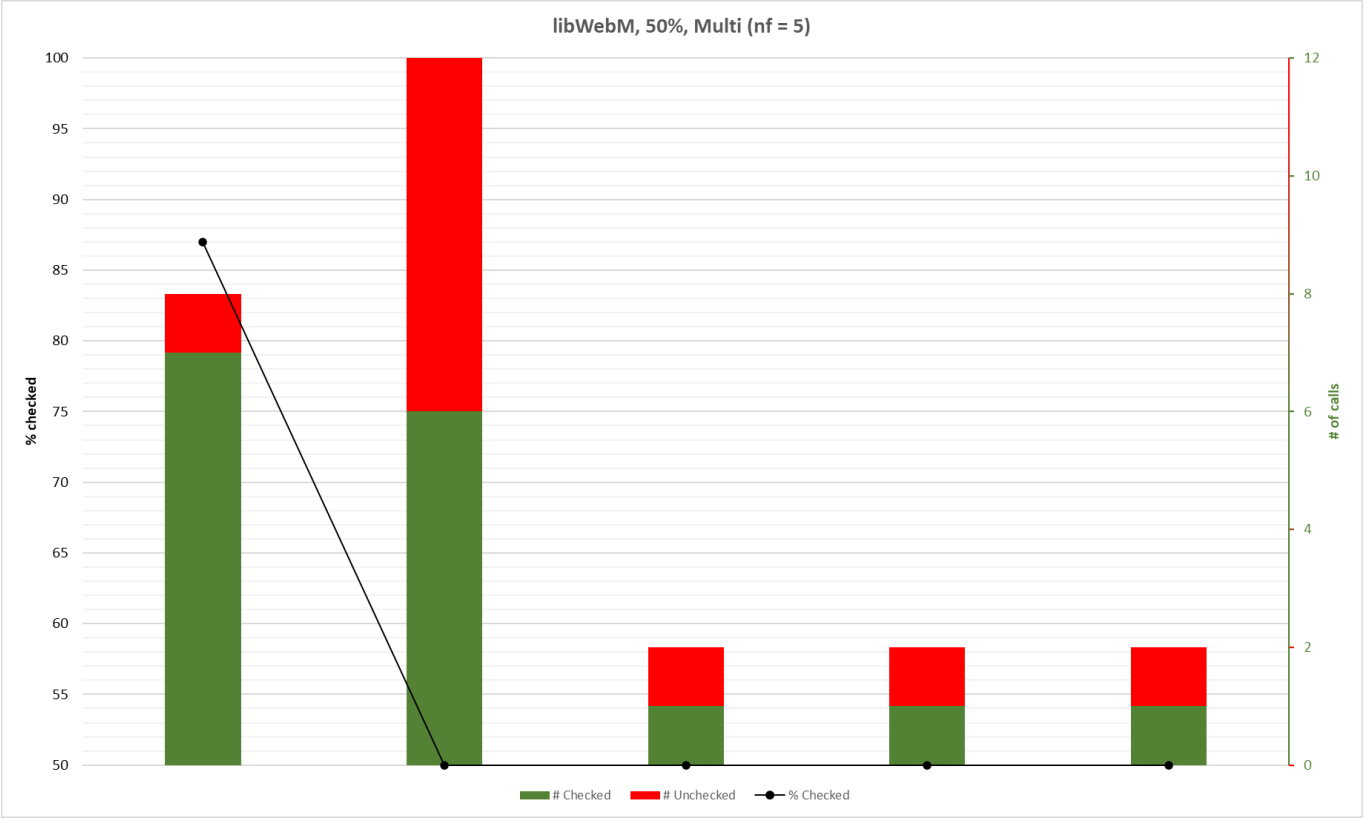

FFmpeg, 50%, Single (nf = 503)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
200
180
160
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked


FFmpeg, 50%, Multi (nf = 297)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
Checked
Unchecked
% Checked

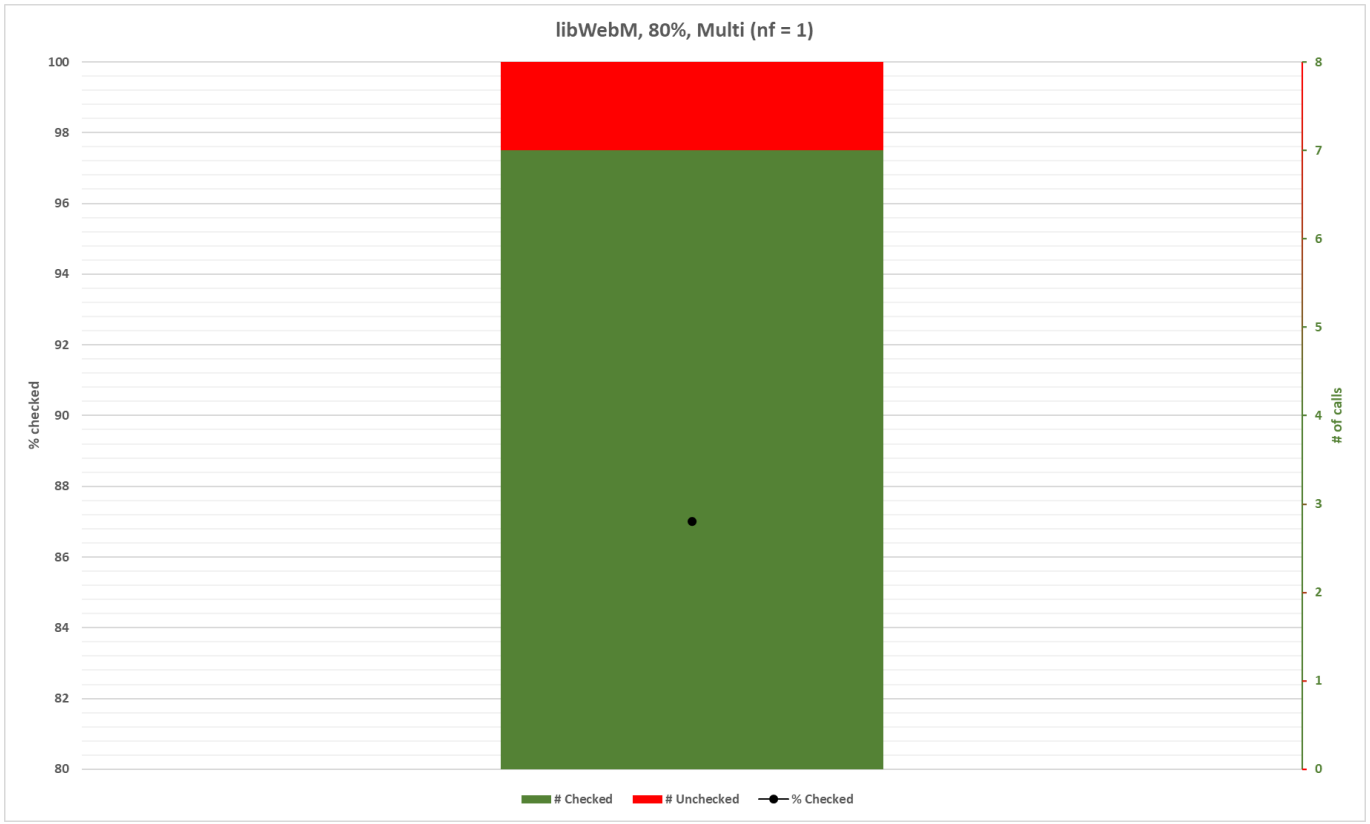

FFmpeg, 80%, Single (nf = 83)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

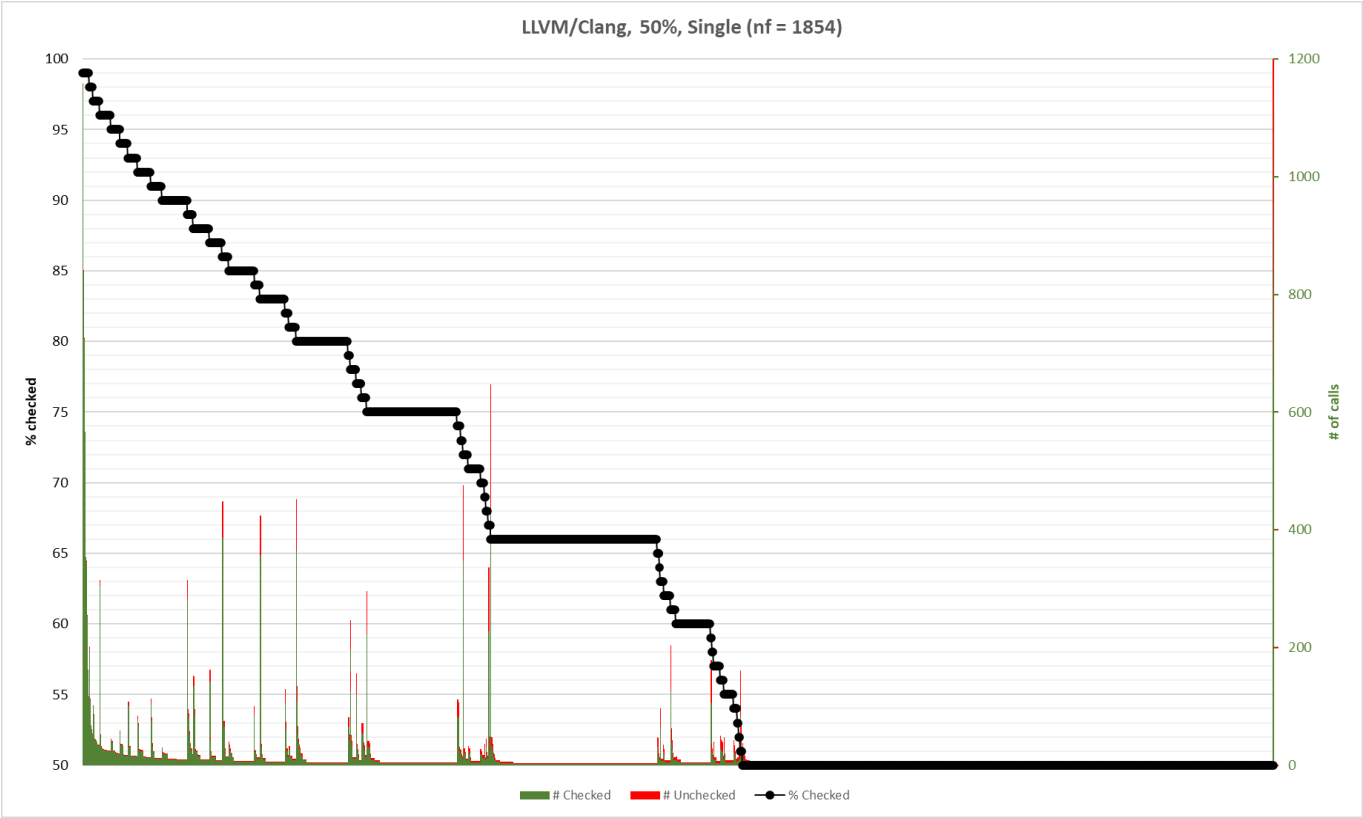

FFmpeg, 80%, Multi (nf = 81)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
Checked
Unchecked
% Checked

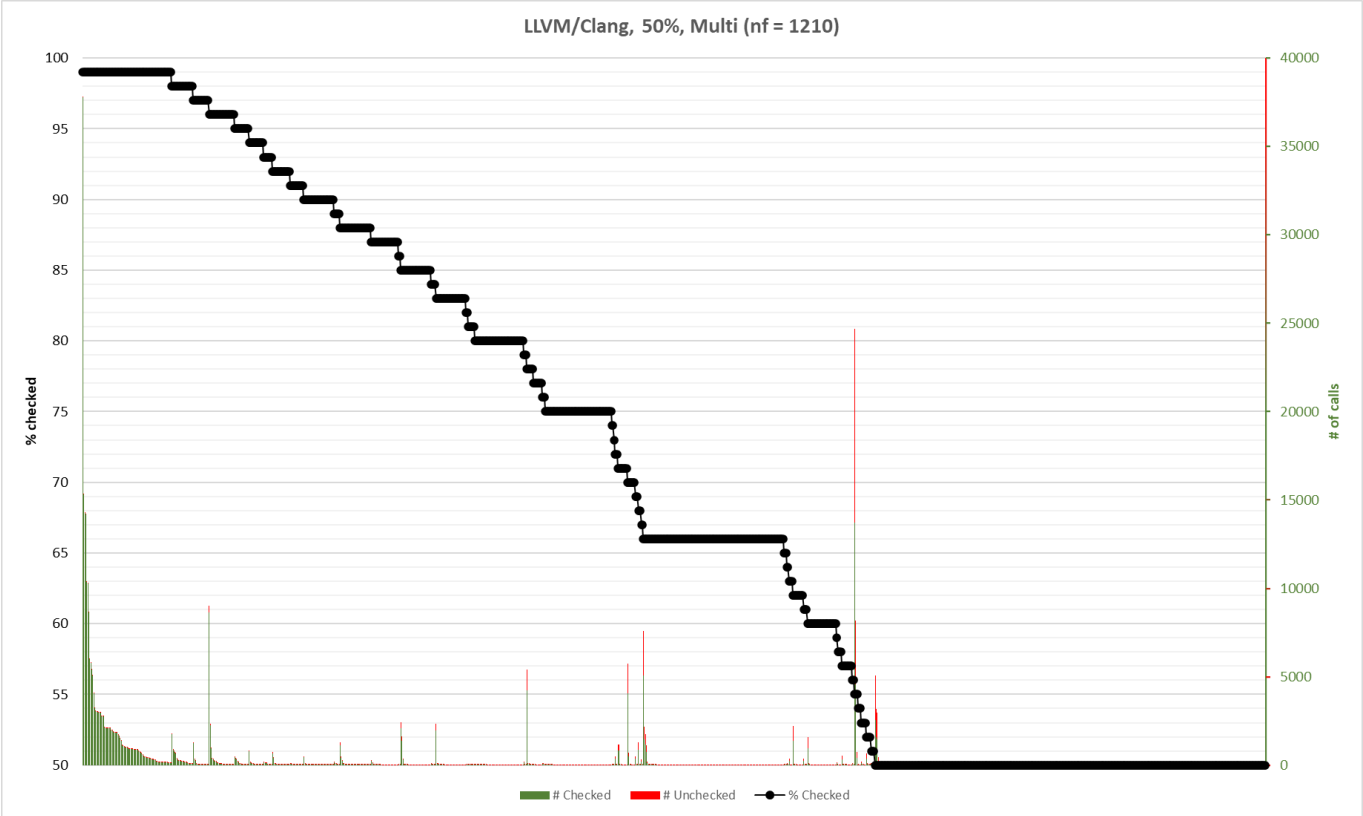

libWebM, 50%, Single (nf = 4)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
12
10
8
6
4
2
0
Checked
Unchecked
% Checked
Series4

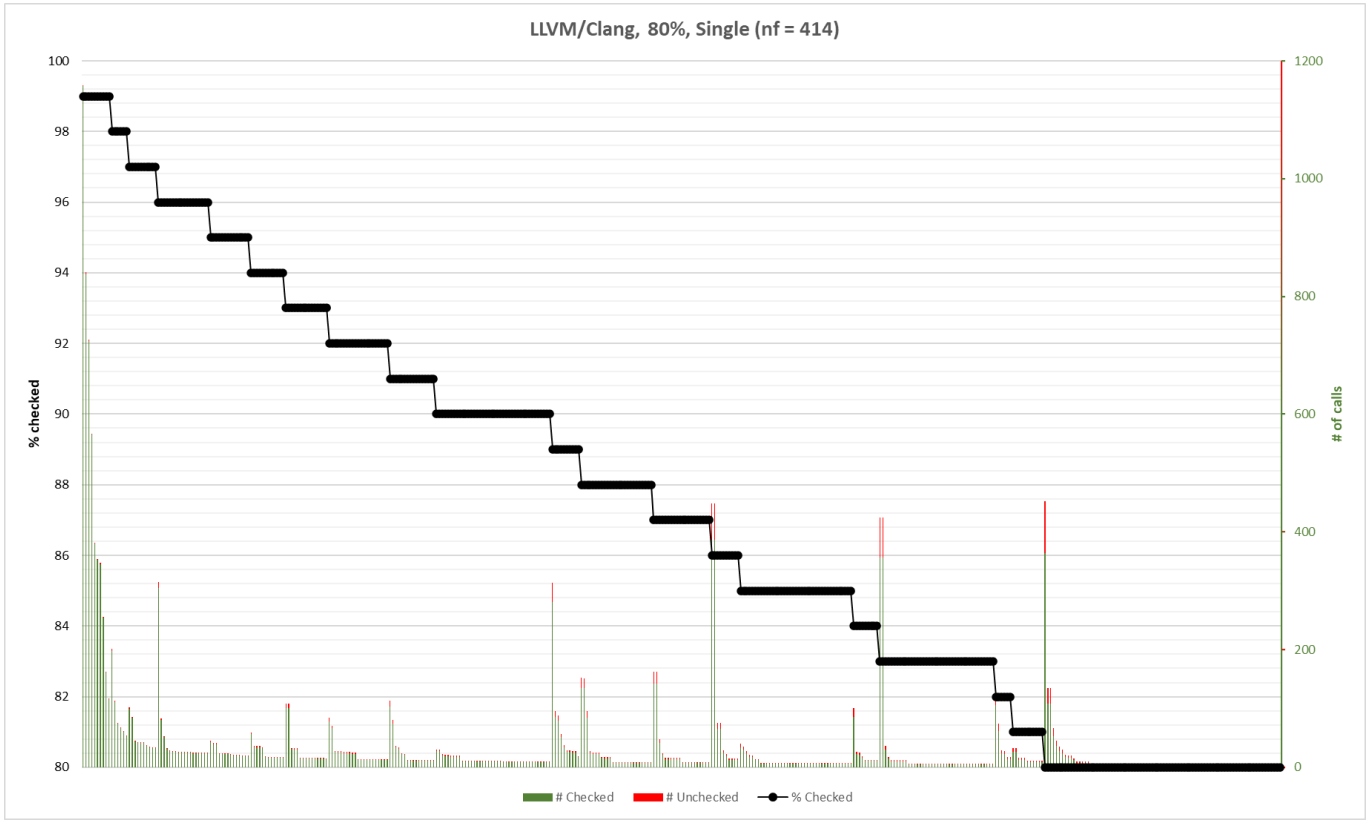

libWebM, 50%, Multi (nf = 5)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
12
10
8
6
4
2
0
Checked
Unchecked
% Checked

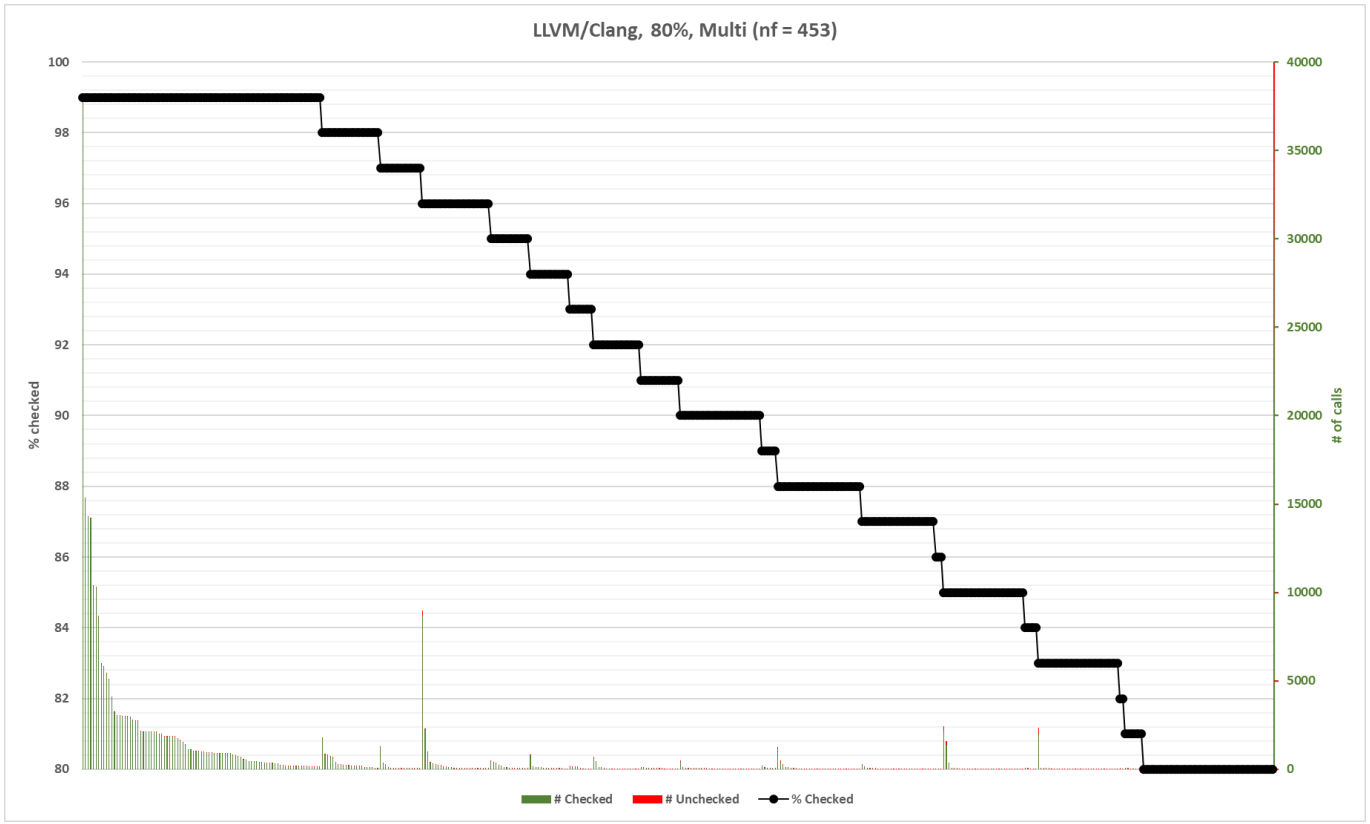

libWebM, 80%, Single (nf = 0)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
1
0,9
0,8
0,7
0,6
0,5
0,4
0,3
0,2
0,1
0
Checked
Unchecked
% Checked

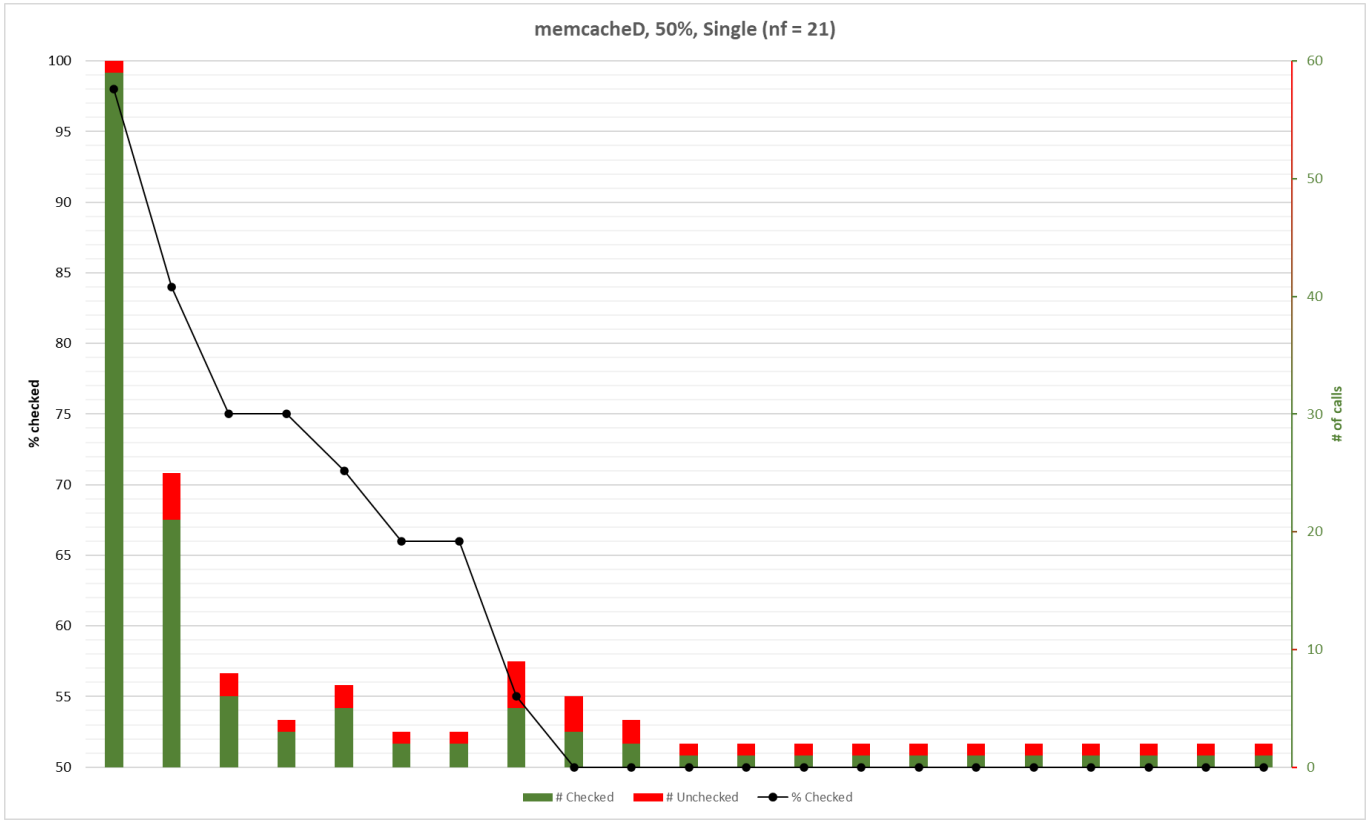

libWebM, 80%, Multi (nf = 1)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
8
7
6
5
4
3
2
1
0
Checked
Unchecked
% Checked

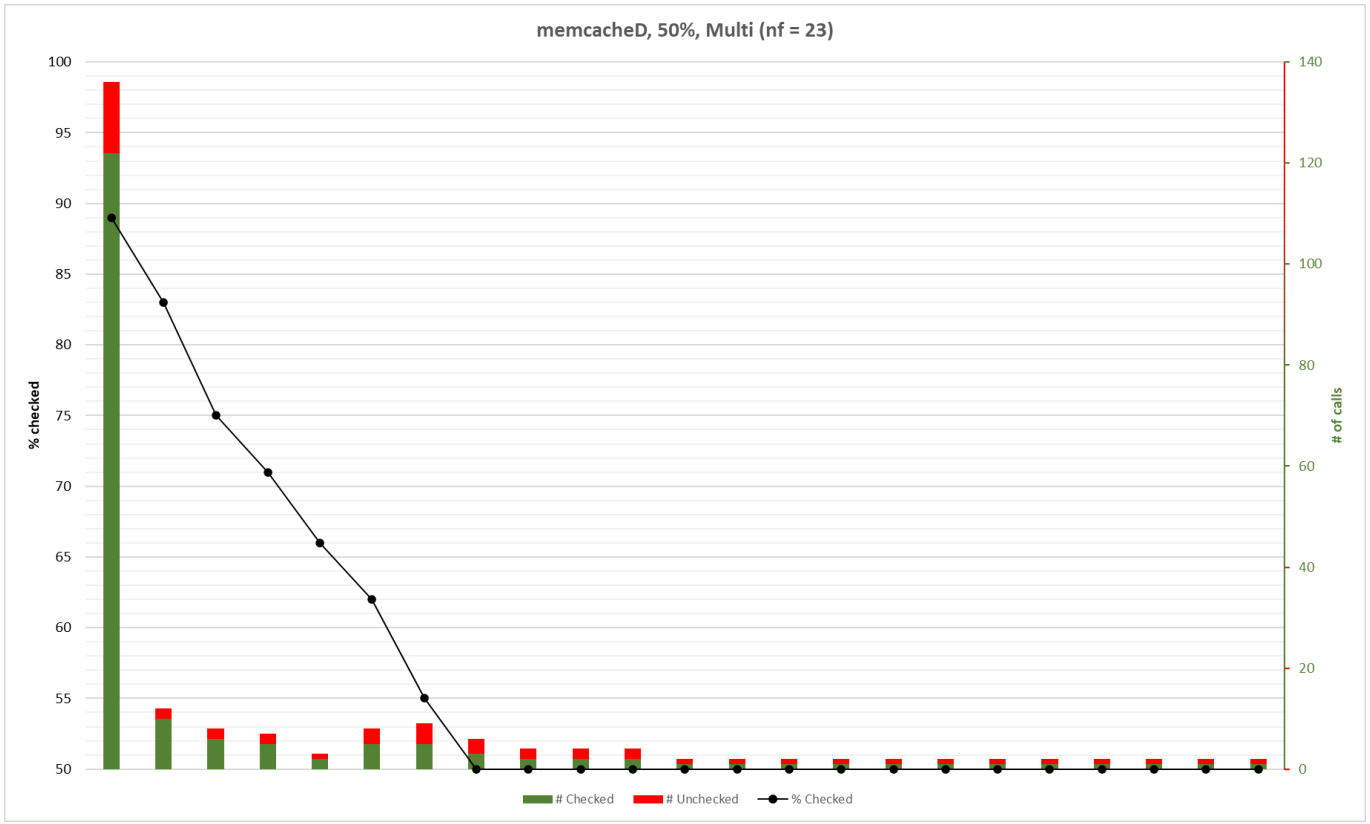

LLVM/Clang, 50%, Single (nf = 1854)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
1200
1000
800
600
400
200
0
Checked
Unchecked
% Checked

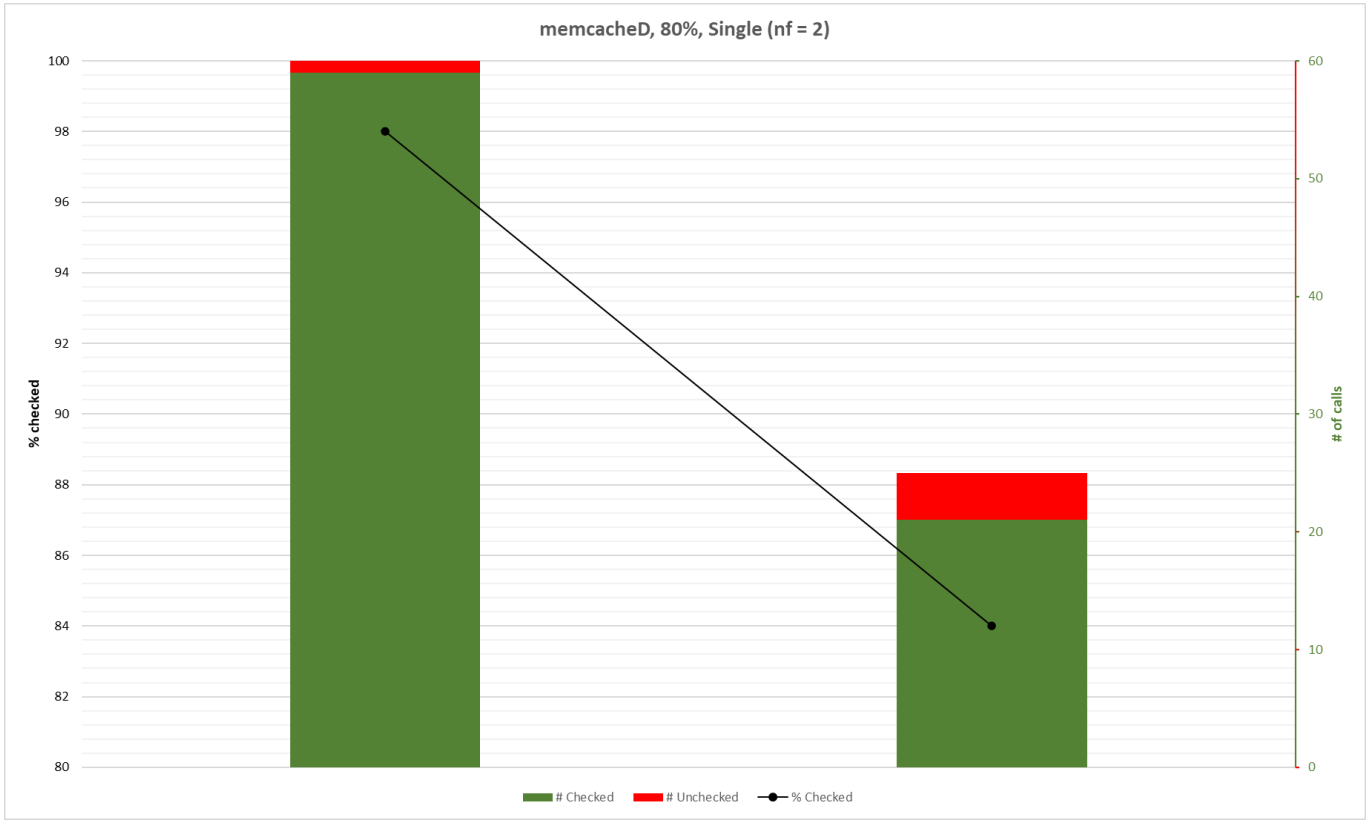

LLVM/Clang, 50%, Multi (nf = 1210)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
40000
35000
30000
25000
20000
15000
10000
5000
0
Checked
Unchecked
% Checked

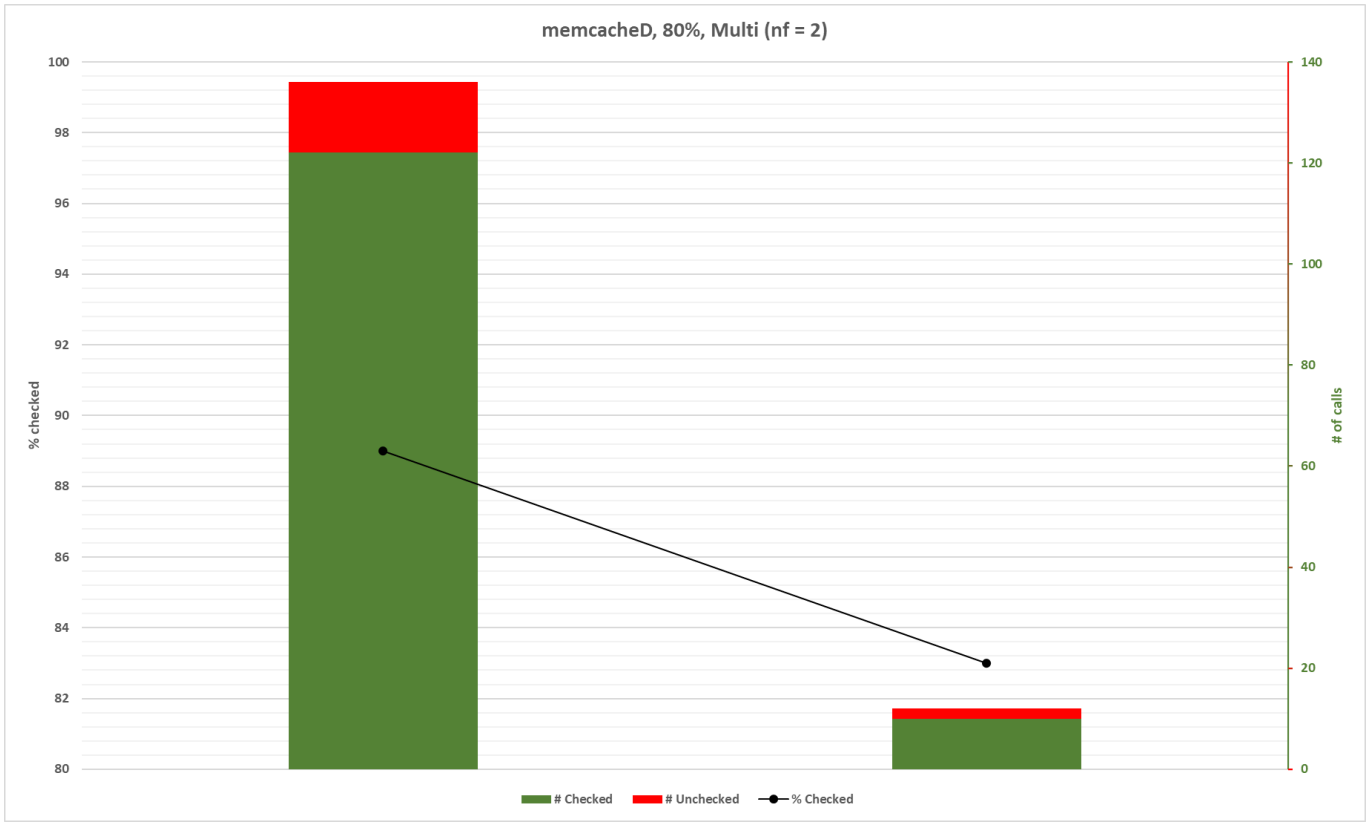

LLVM/Clang, 80%, Single (nf = 414)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
1200
1000
800
600
400
200
0
Checked
Unchecked
% Checked

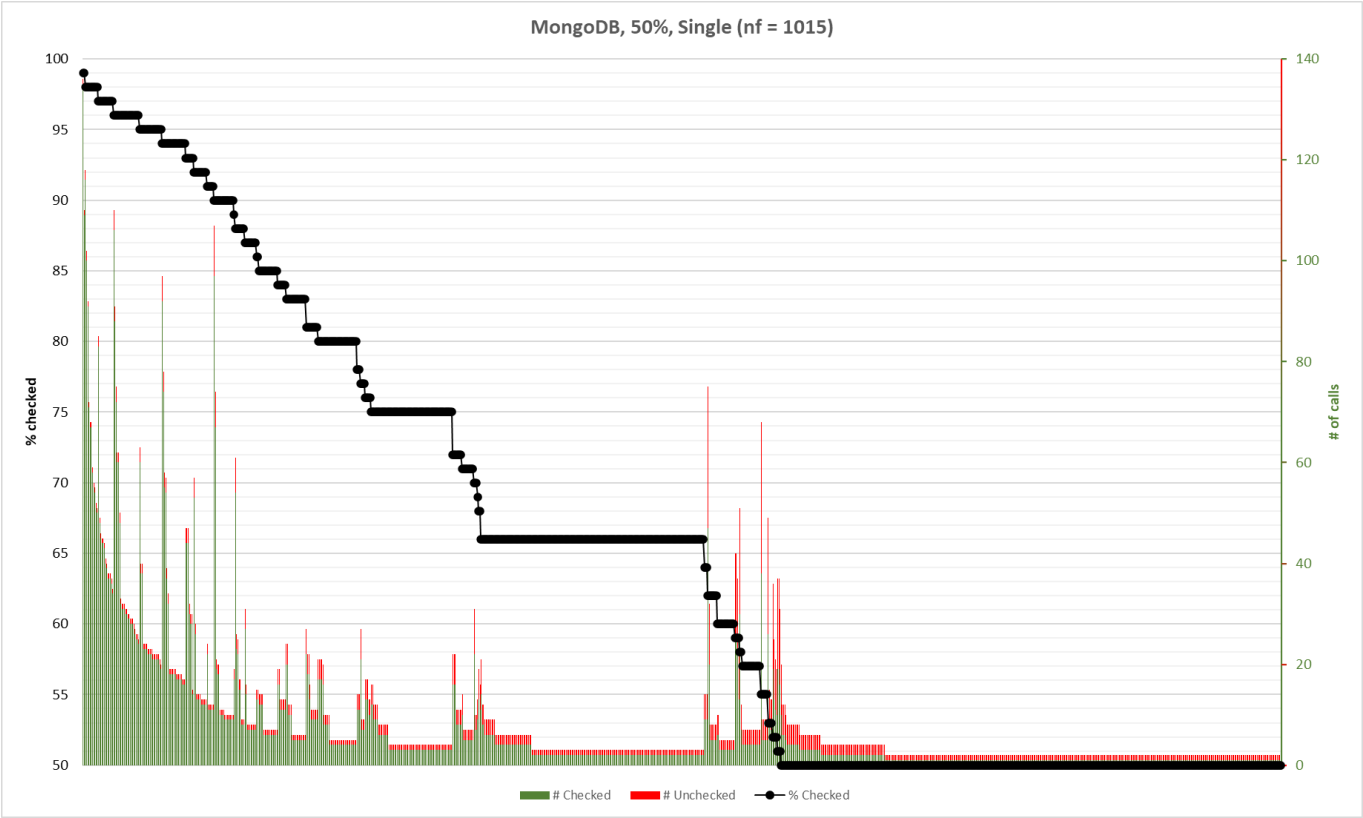

LLVM/Clang, 80%, Multi (nf = 453)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
40000
35000
30000
25000
20000
15000
10000
5000
0
Checked
Unchecked
% Checked

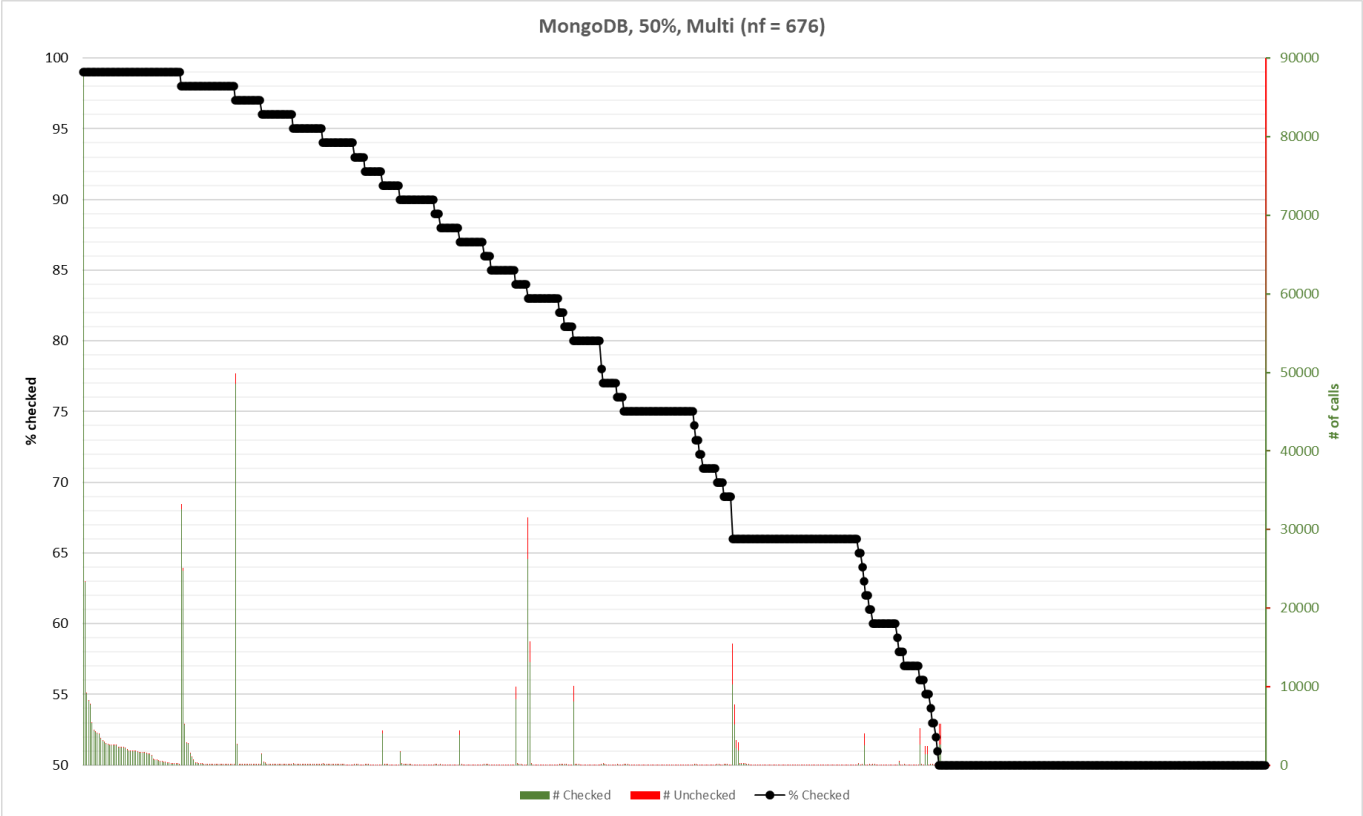

memcacheD, 50%, Single (nf = 21)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
60
50
40
30
20
10
0
Checked
Unchecked
% Checked

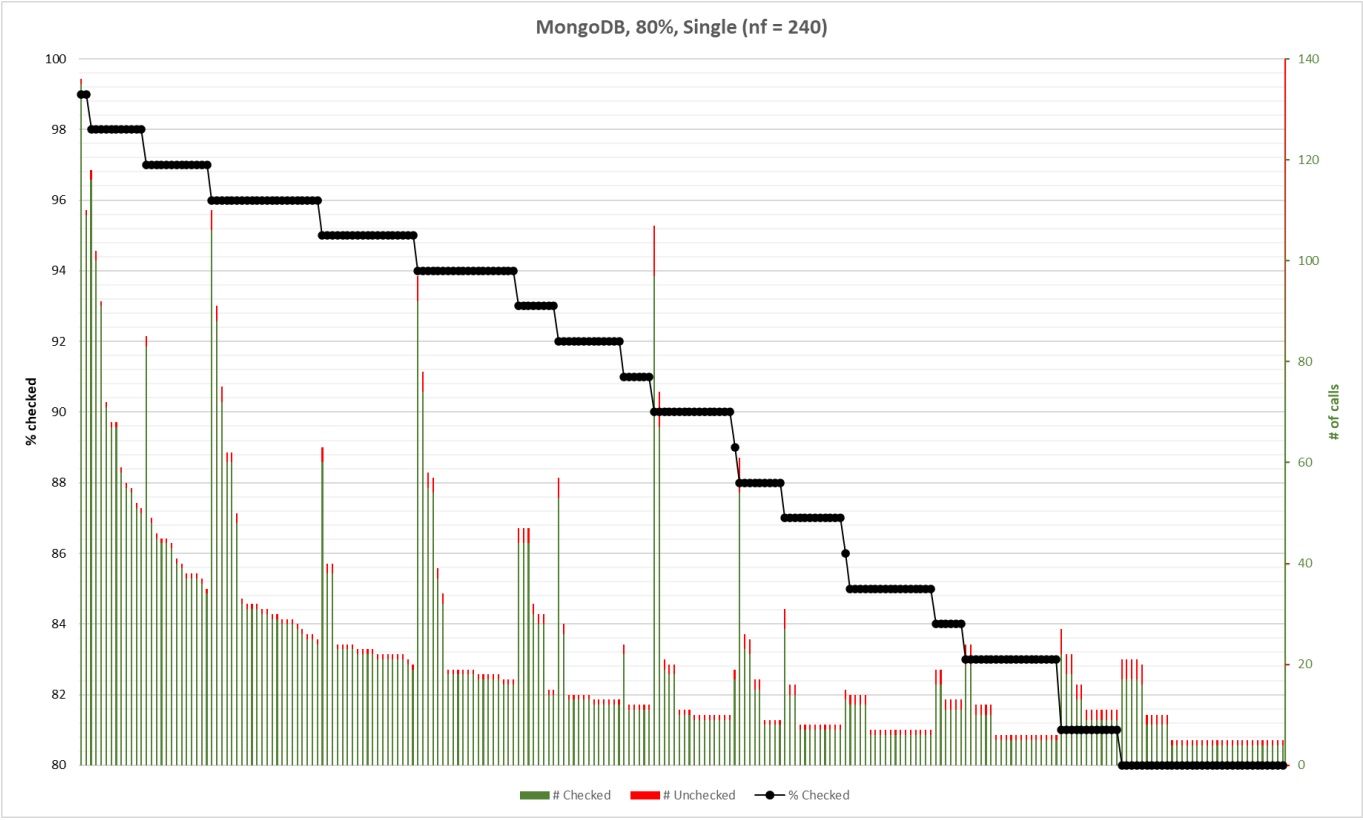

memcacheD, 50%, Multi (nf = 23)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

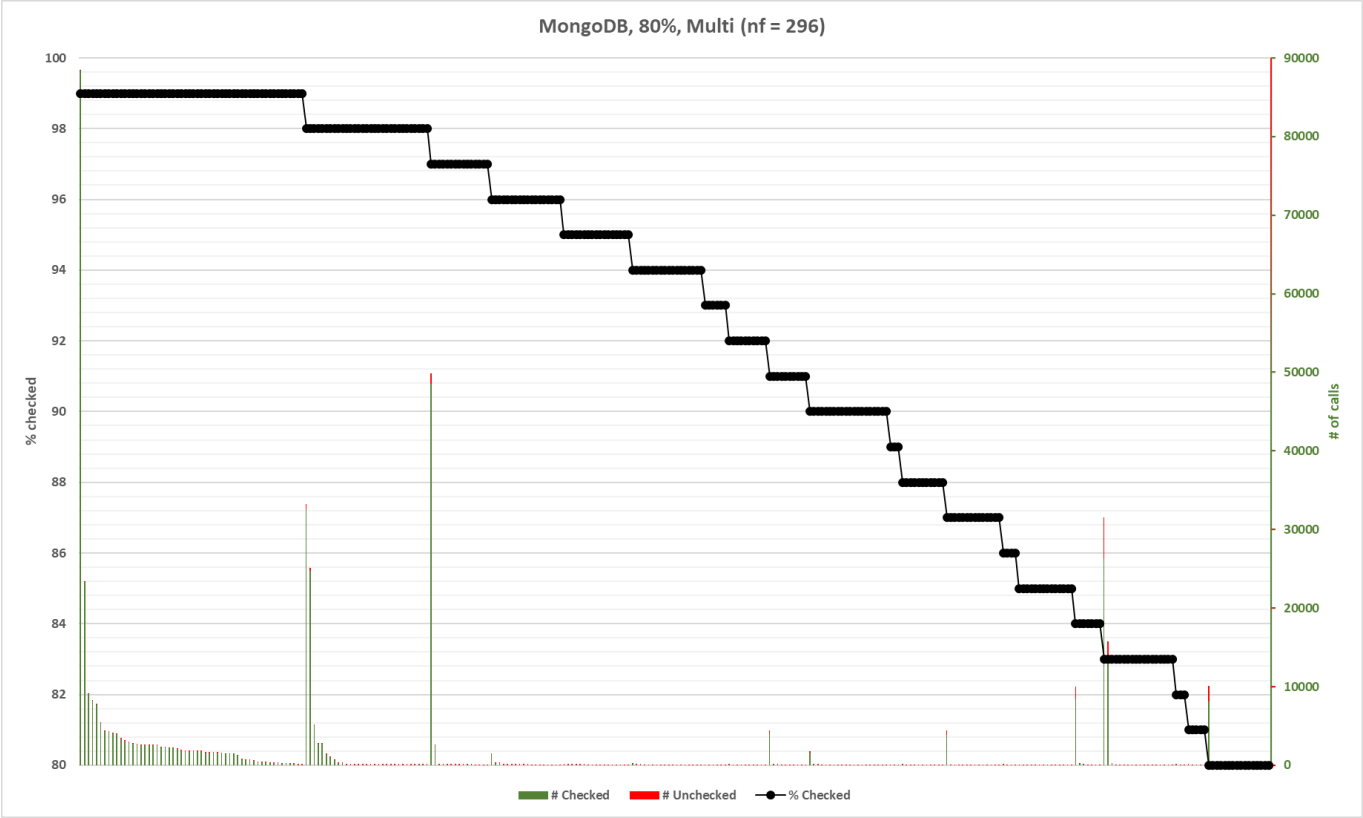

memcacheD, 80%, Single (nf = 2)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
60
50
40
30
20
10
0
Checked
Unchecked
% Checked

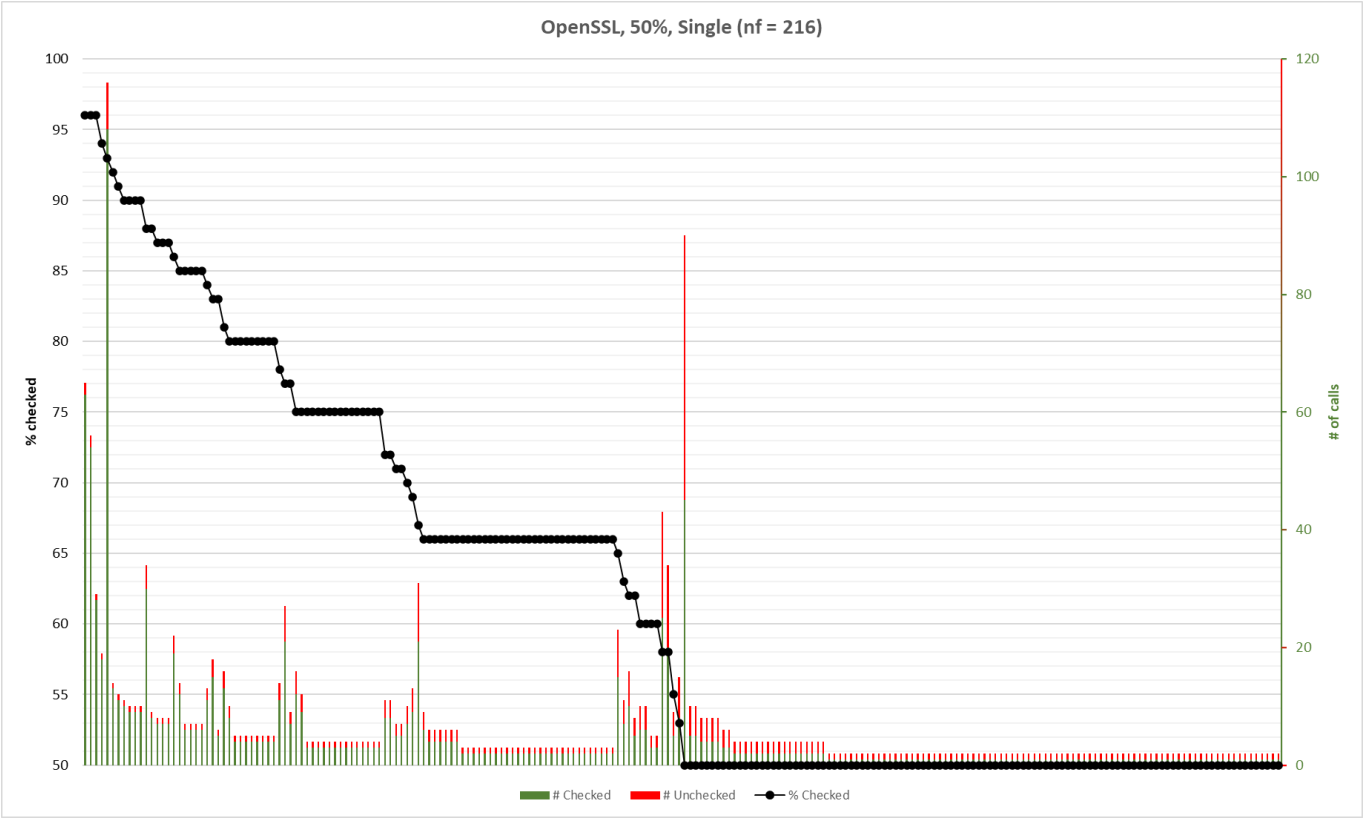

memcacheD, 80%, Multi (nf = 2)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

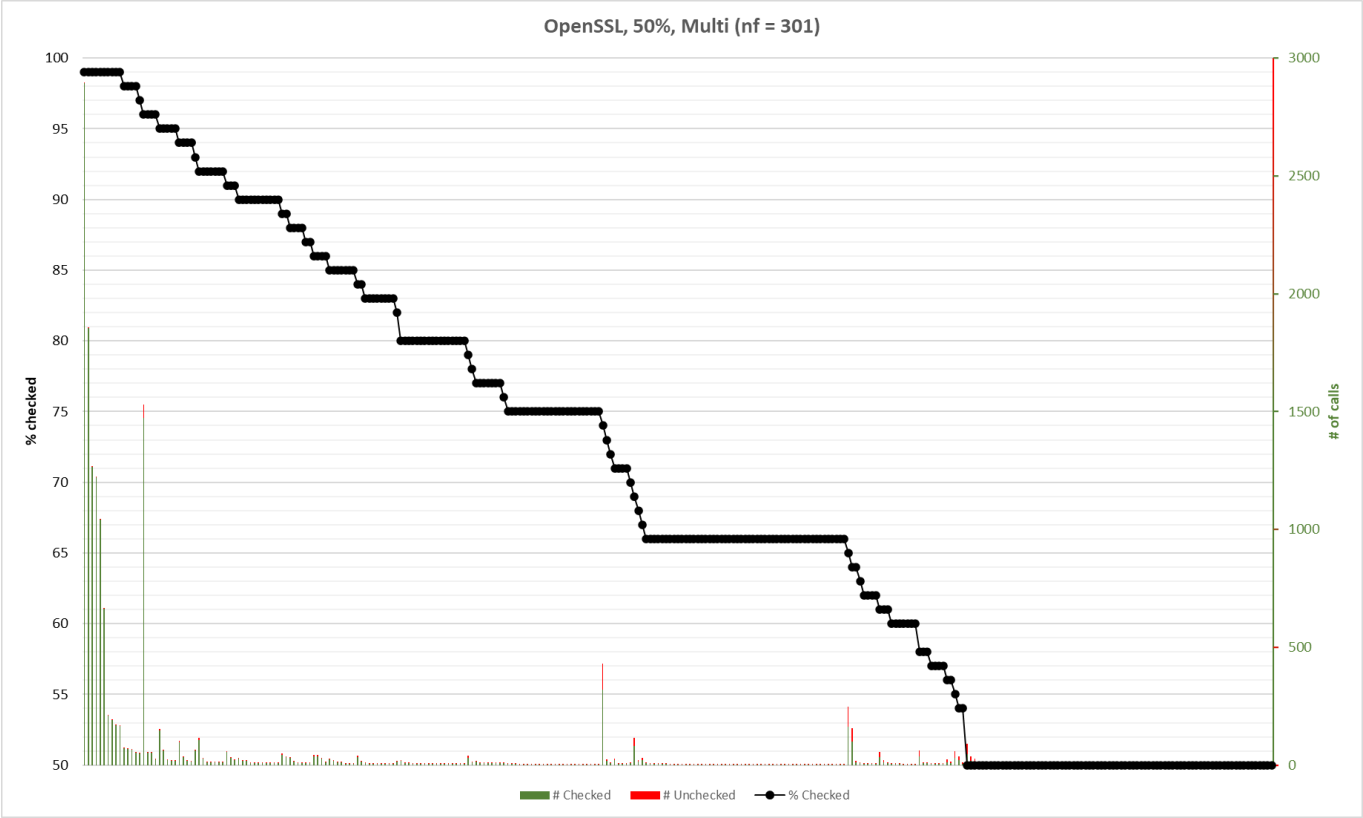

MongoDB, 50%, Single (nf = 1015)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

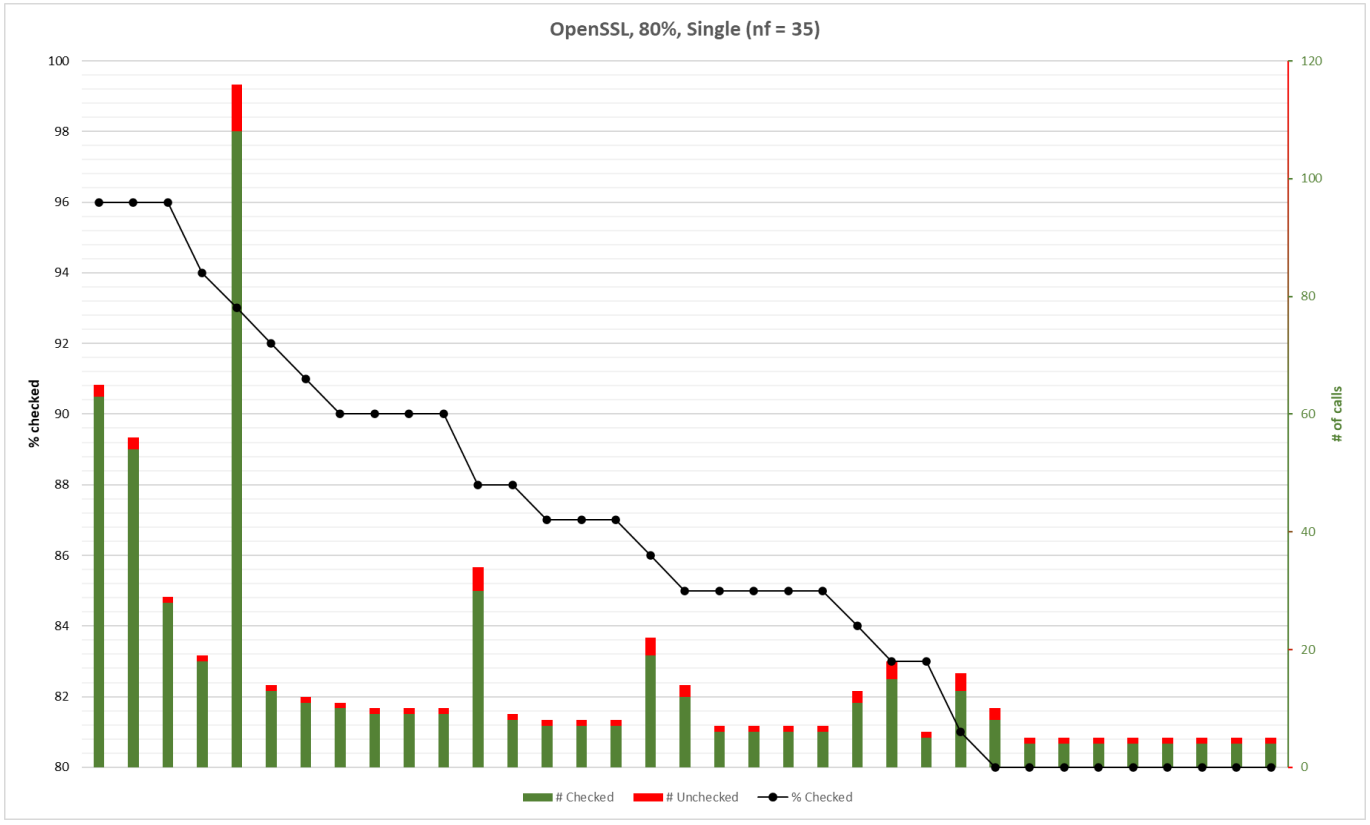

MongoDB, 50%, Multi (nf = 676)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Checked
Unchecked
% Checked

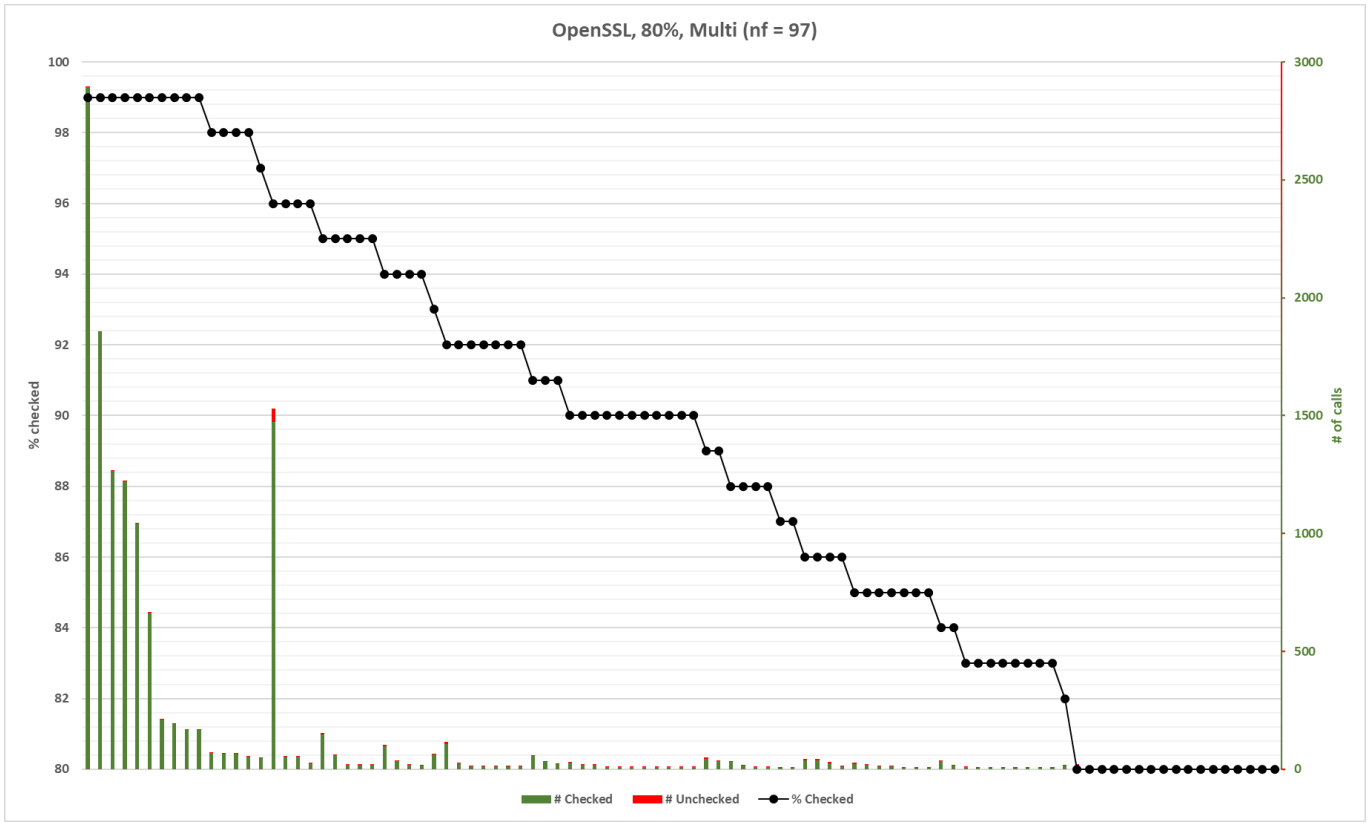

MongoDB, 80%, Single (nf = 240)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

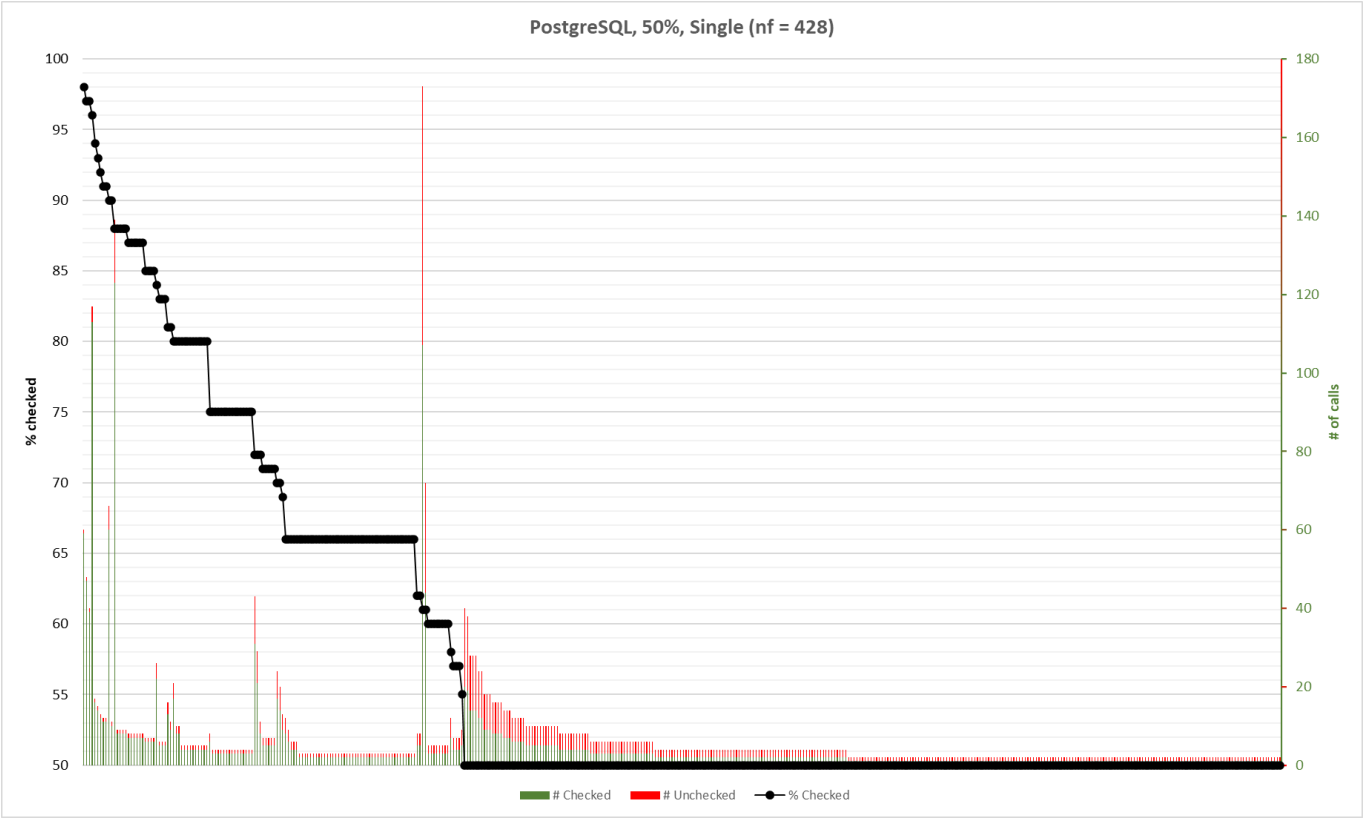

MongoDB, 80%, Multi (nf = 296)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Checked
Unchecked
% Checked

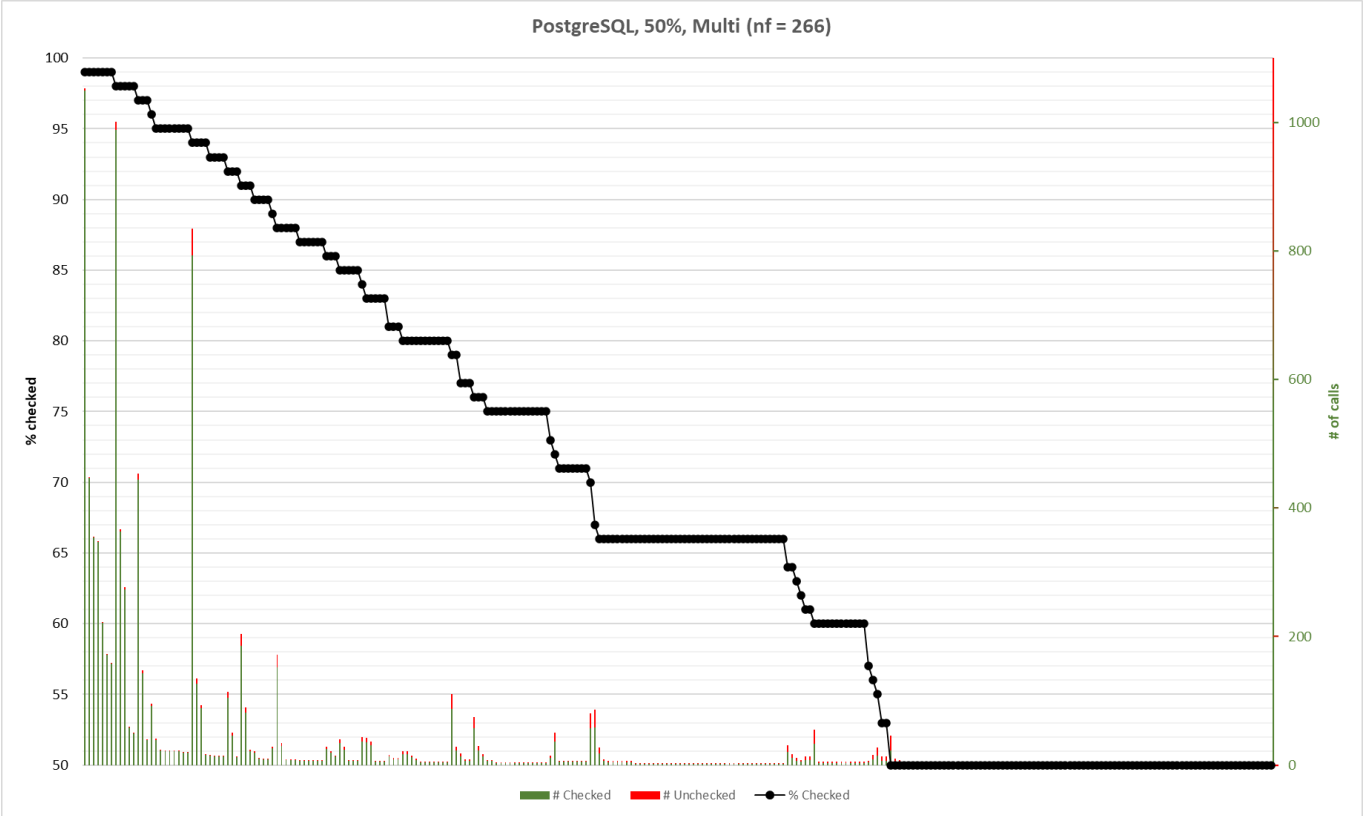

OpenSSL, 50%, Single (nf = 216)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

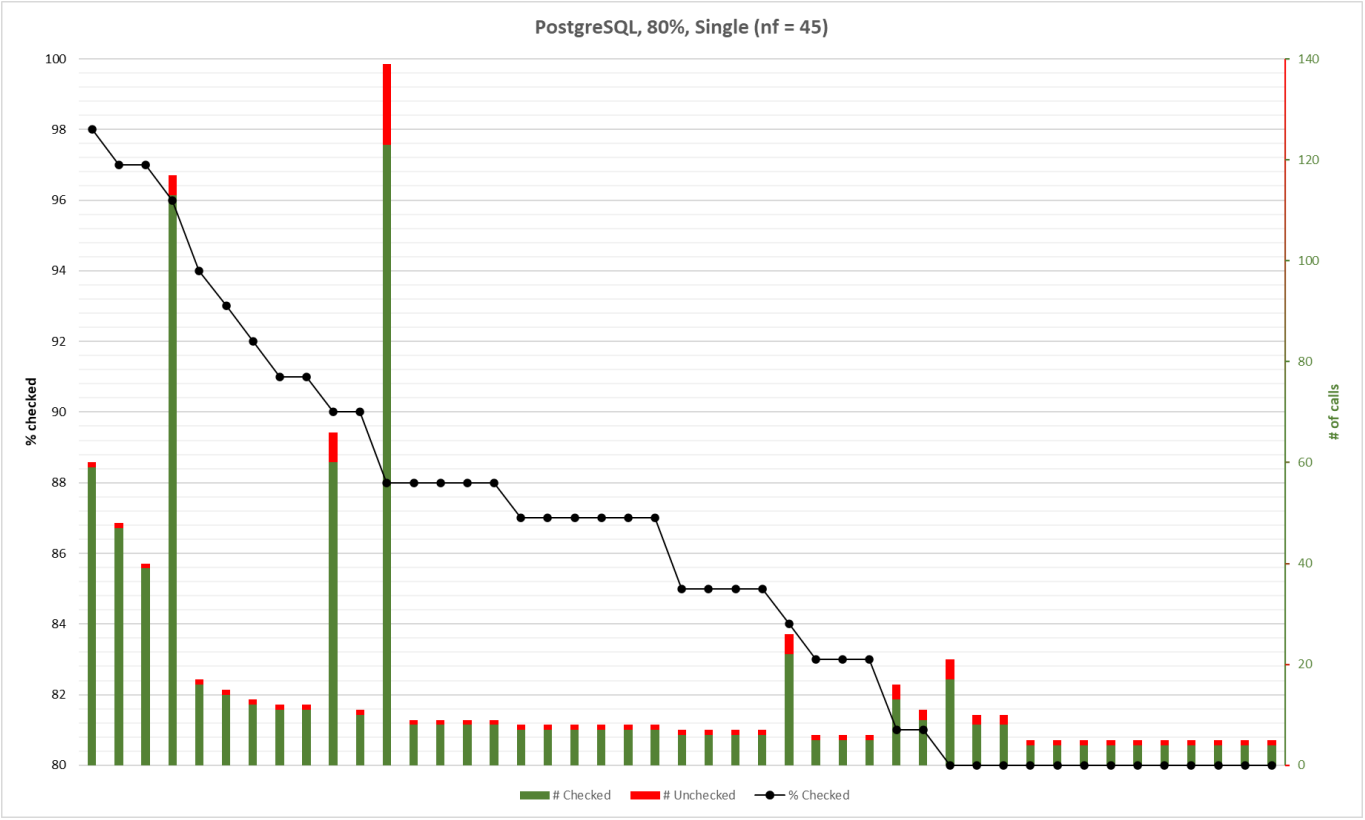

OpenSSL, 50%, Multi (nf = 301)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
3000
2500
2000
1500
1000
500
0
Checked
Unchecked
% Checked

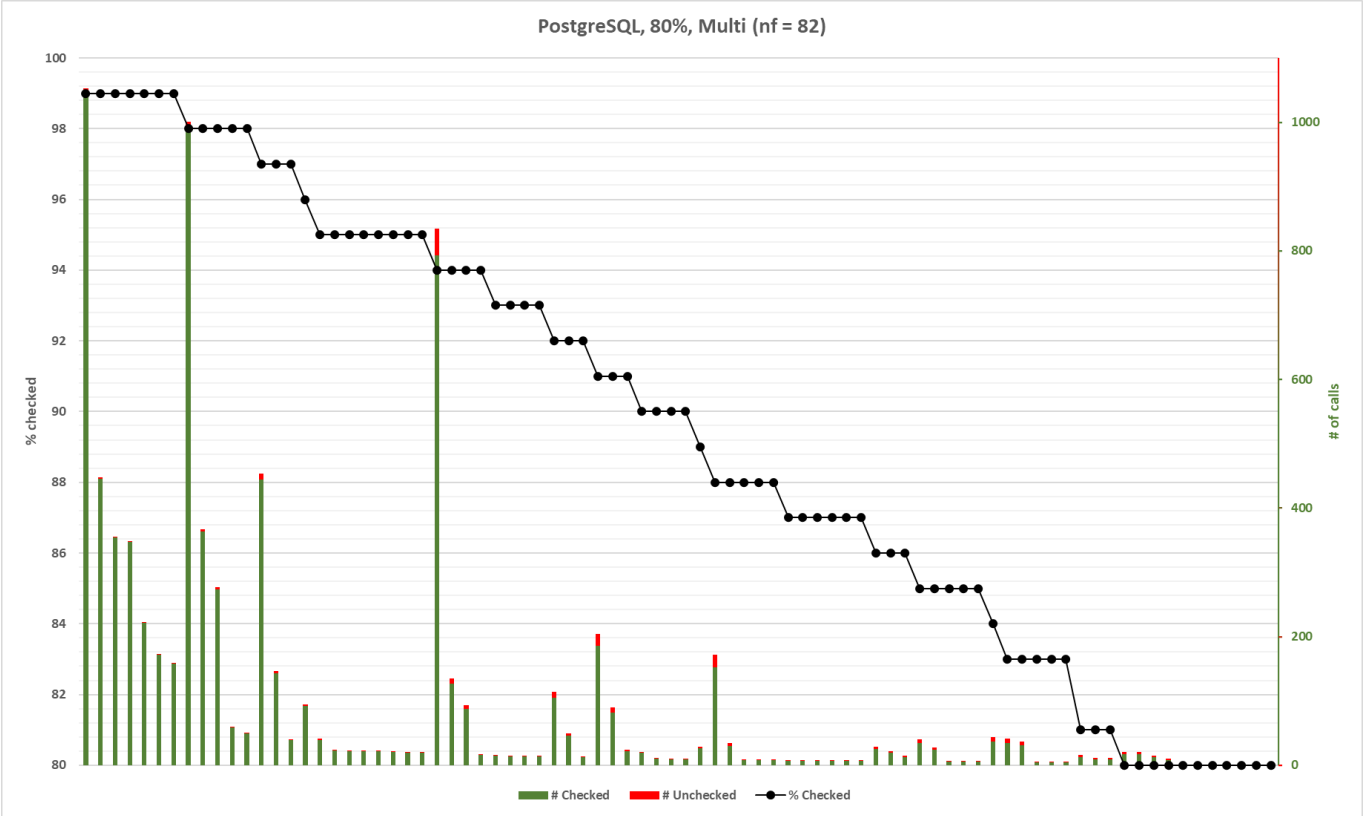

OpenSSL, 80%, Single (nf = 35)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

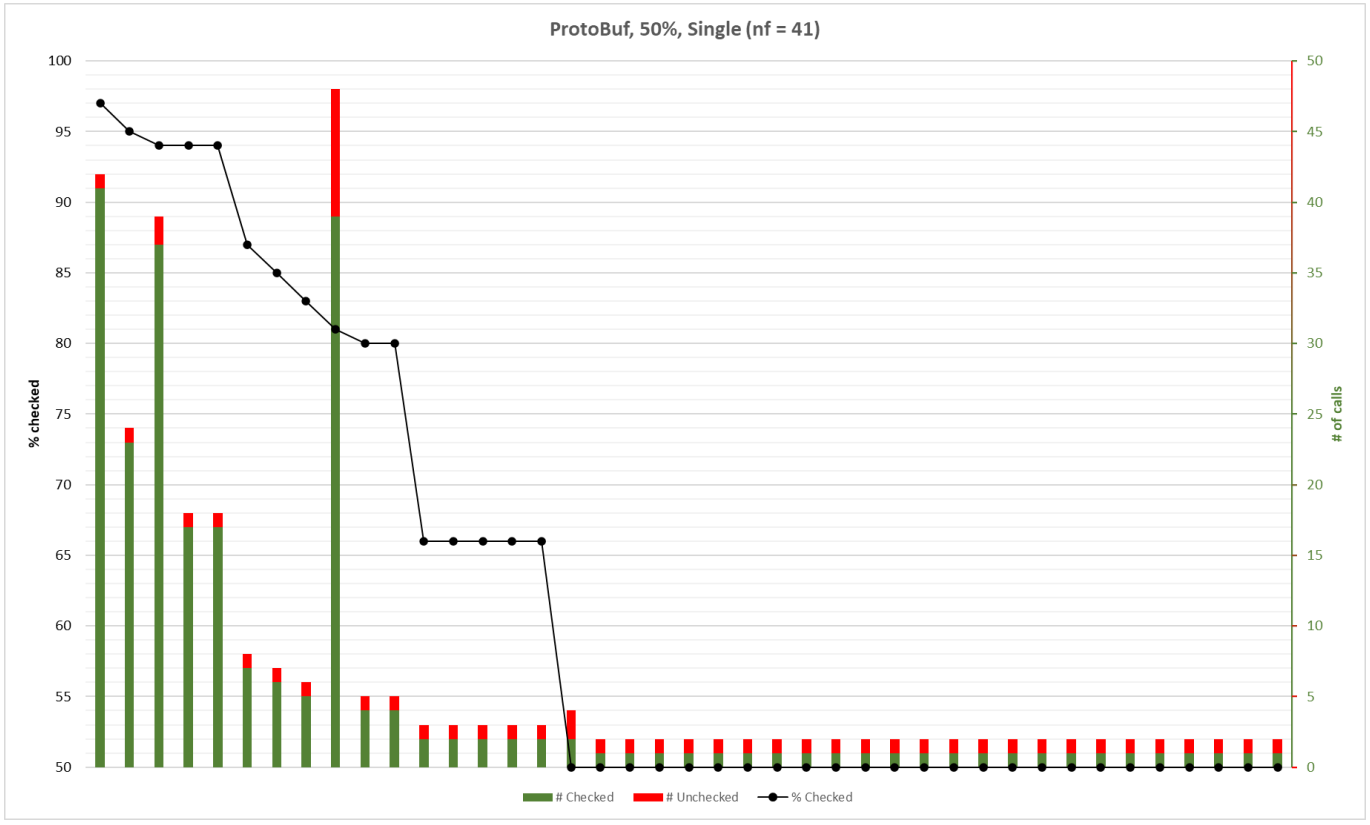

OpenSSL, 80%, Multi (nf = 97)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
3000
2500
2000
1500
1000
500
0
Checked
Unchecked
% Checked

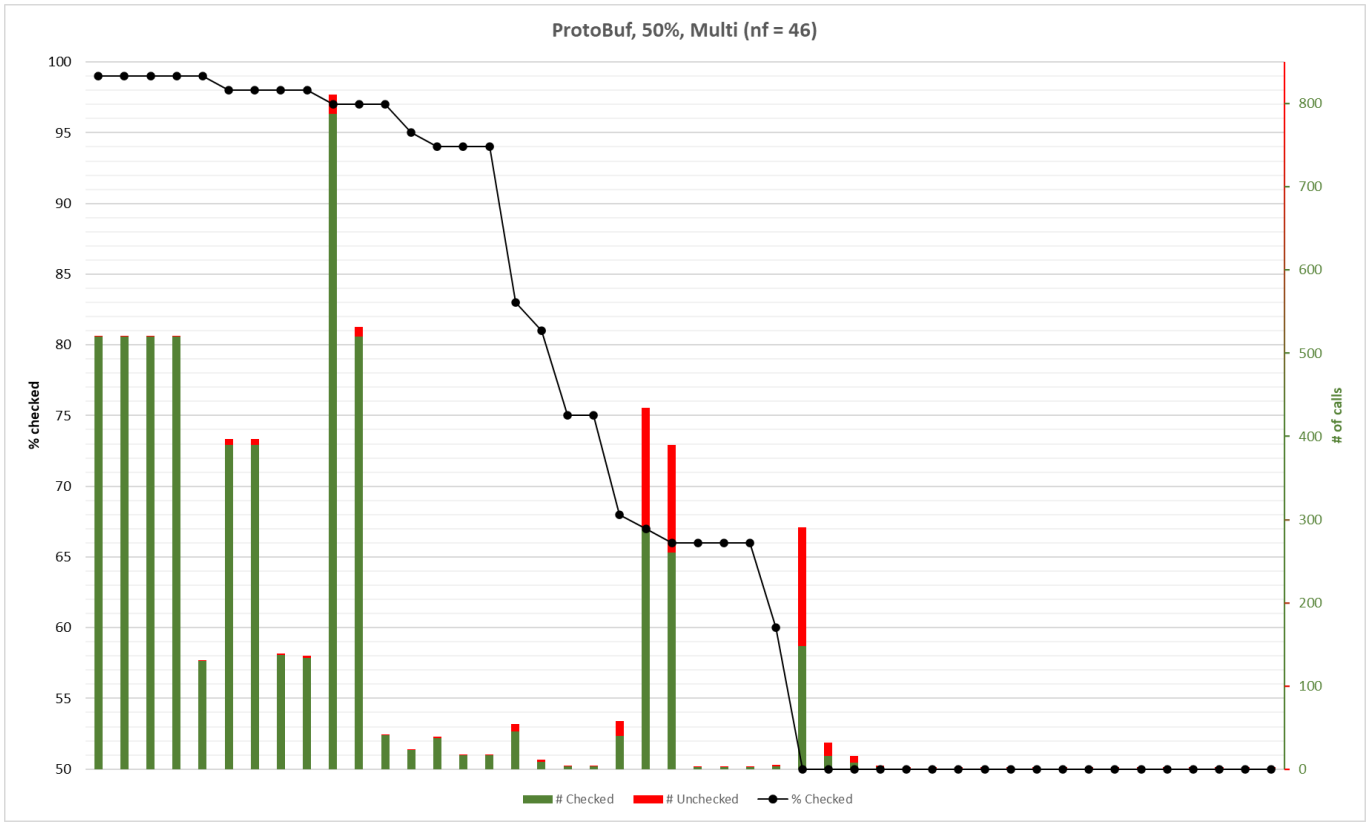

PostgreSQL, 50%, Single (nf = 428)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
180
160
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

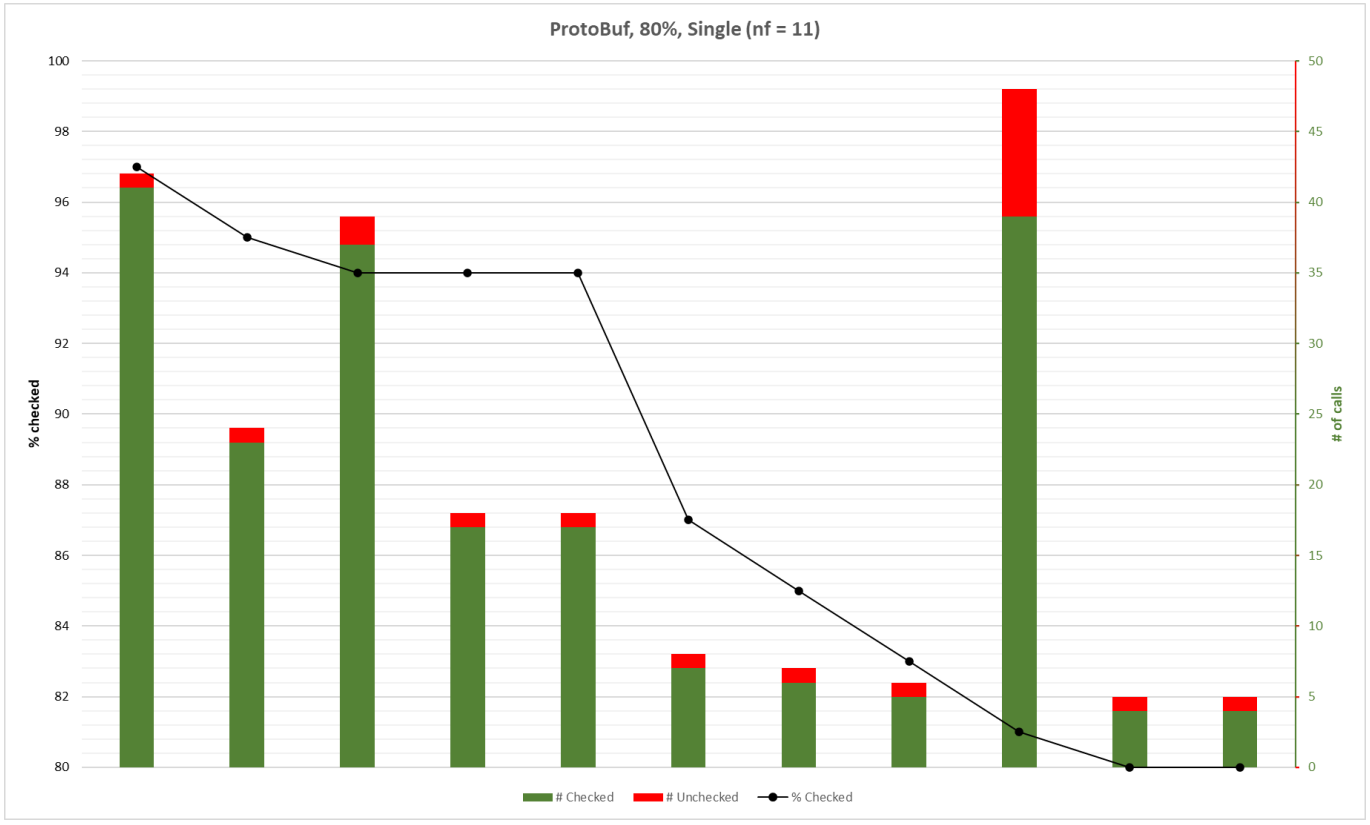

PostgreSQL, 50%, Multi (nf = 266)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
1000
800
600
400
200
0
Checked
Unchecked
% Checked

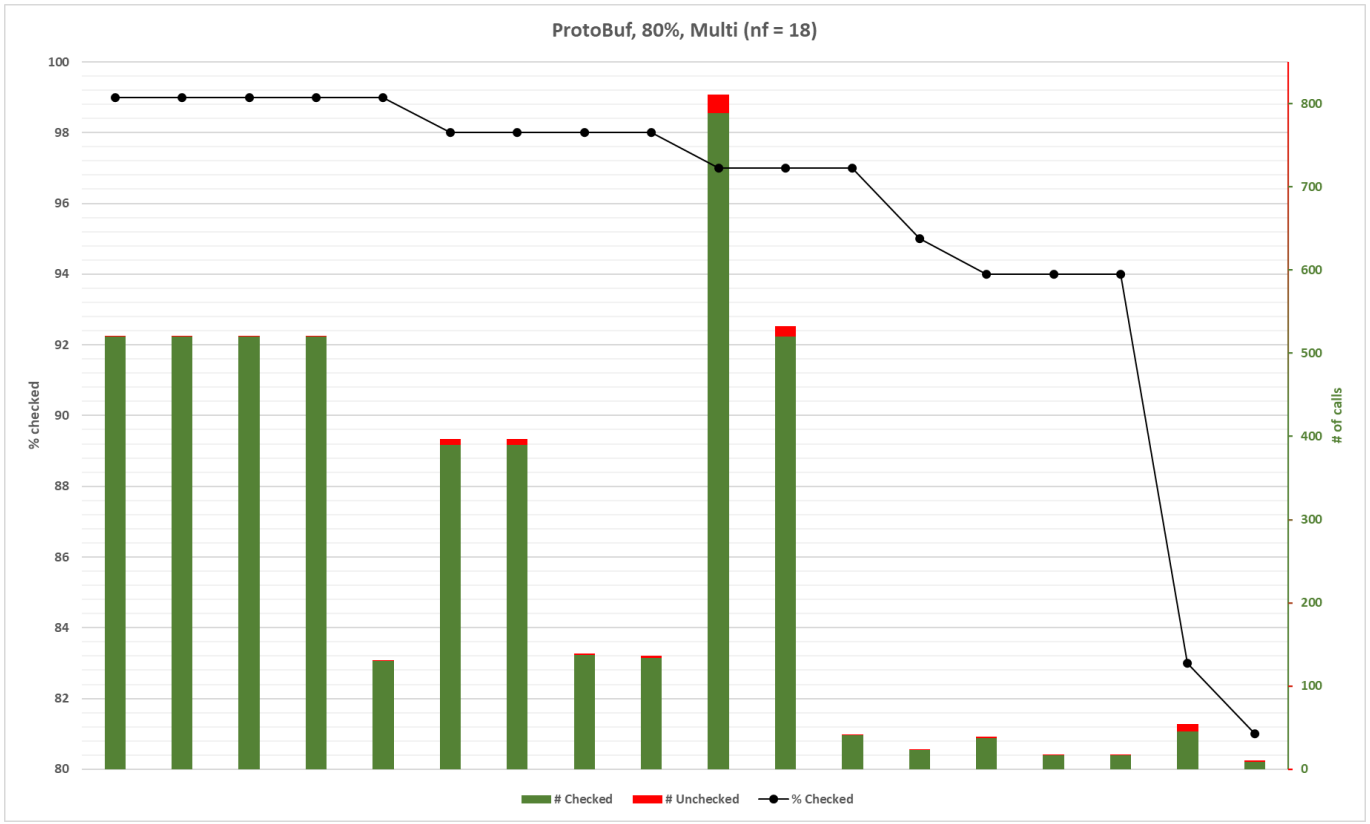

PostgreSQL, 80%, Single (nf = 45)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

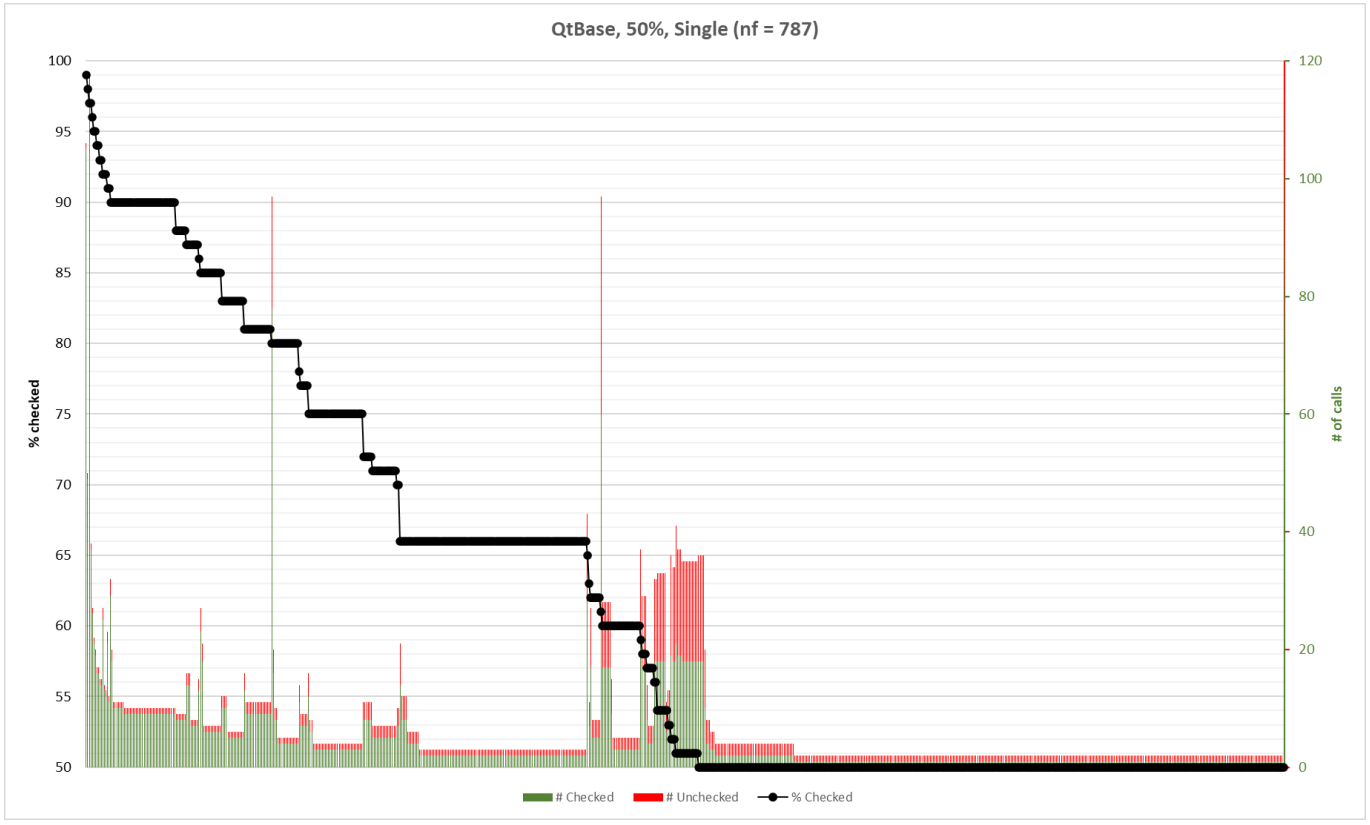

PostgreSQL, 80%, Multi (nf = 82)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
1000
800
600
400
200
0
Checked
Unchecked
% Checked

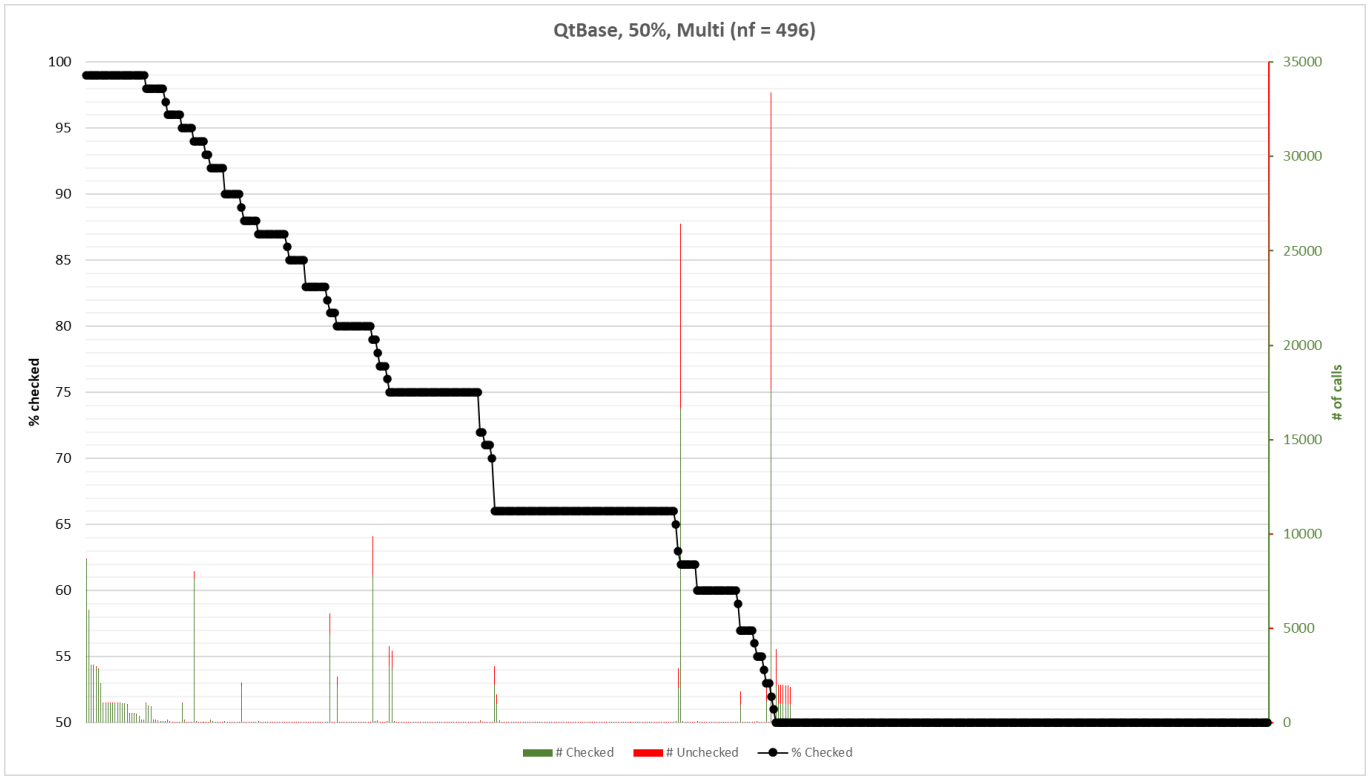

ProtoBuf, 50%, Single (nf = 41)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
50
45
40
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked

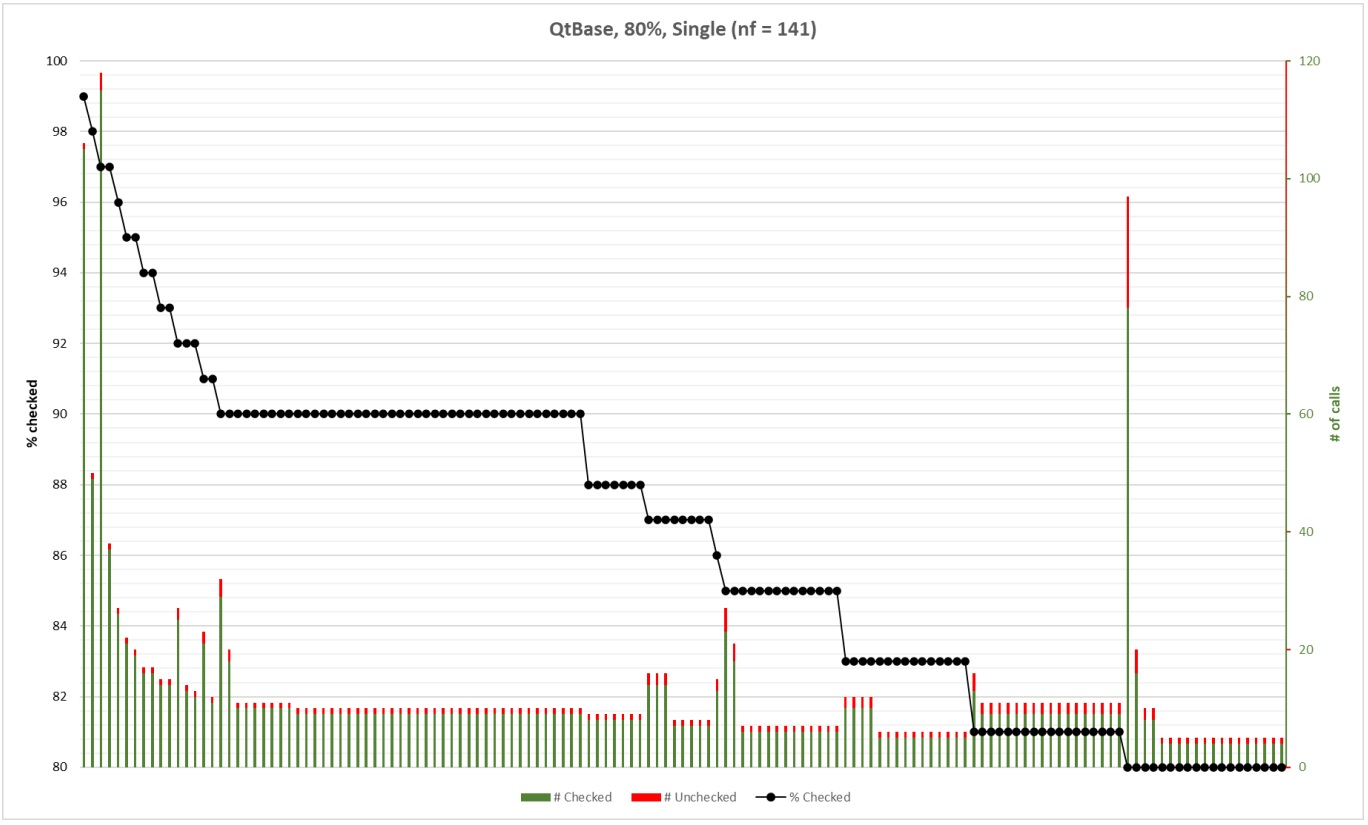

ProtoBuf, 50%, Multi (nf = 46)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
800
700
600
500
400
300
200
100
0
Checked
Unchecked
% Checked

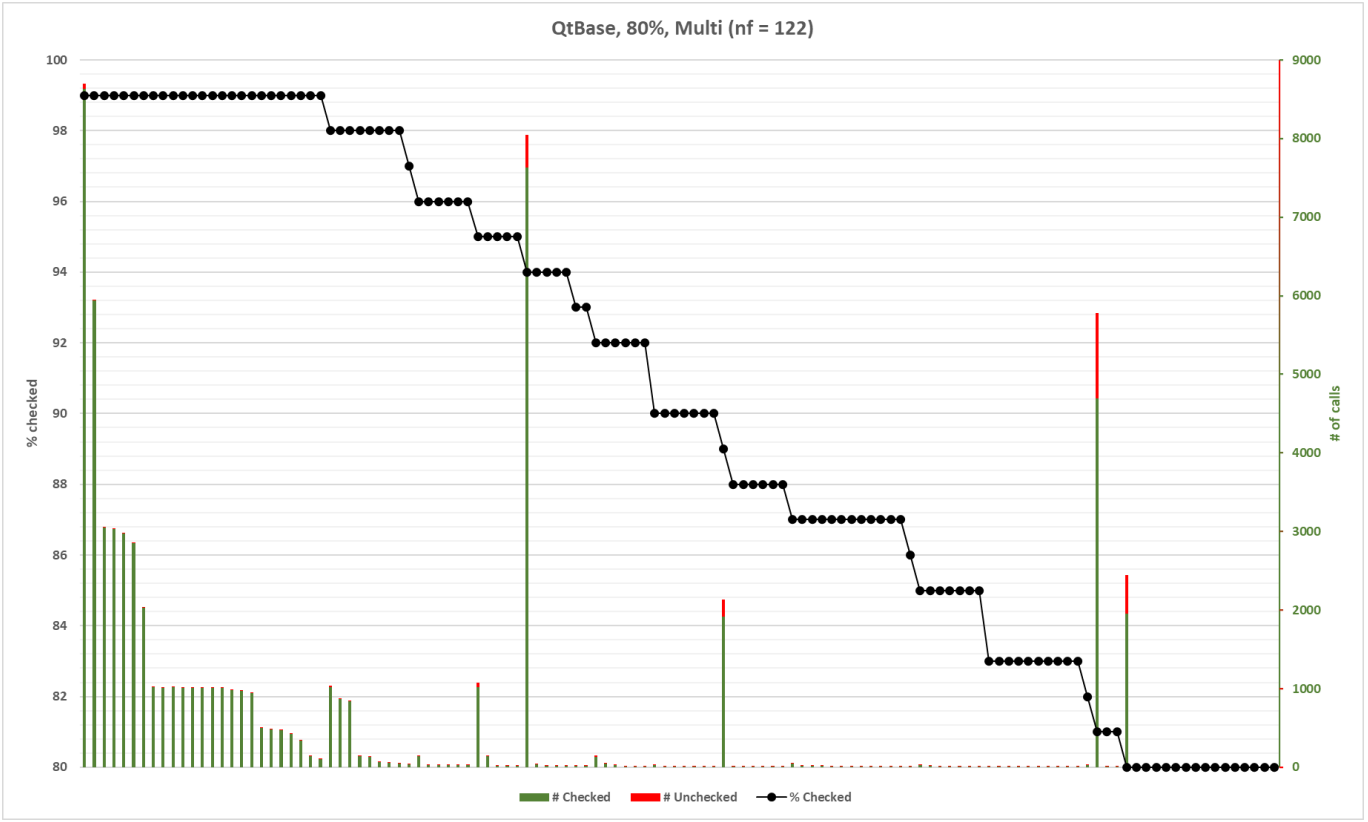

ProtoBuf, 80%, Single (nf = 11)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
50
45
40
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked

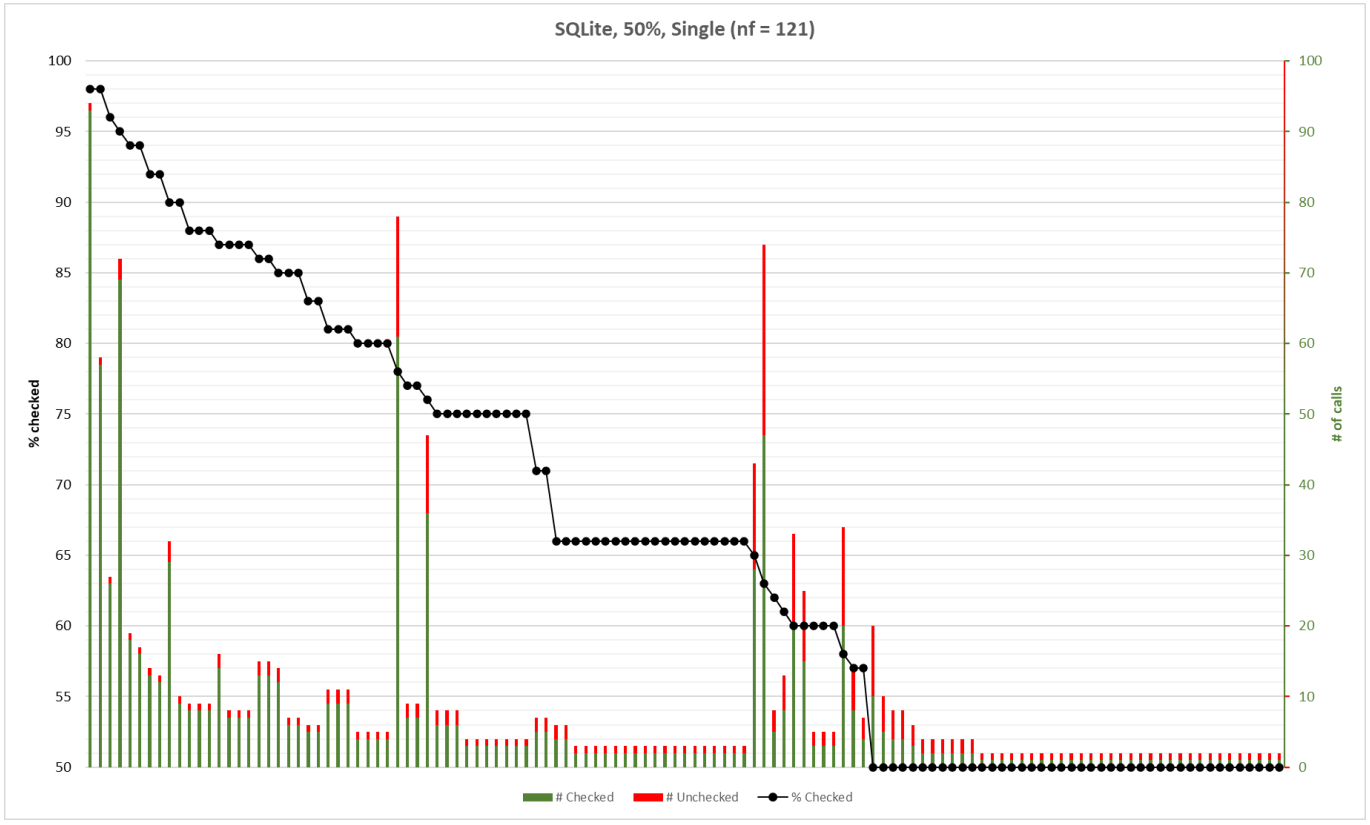

ProtoBuf, 80%, Multi (nf = 18)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
800
700
600
500
400
300
200
100
0
Checked
Unchecked
% Checked

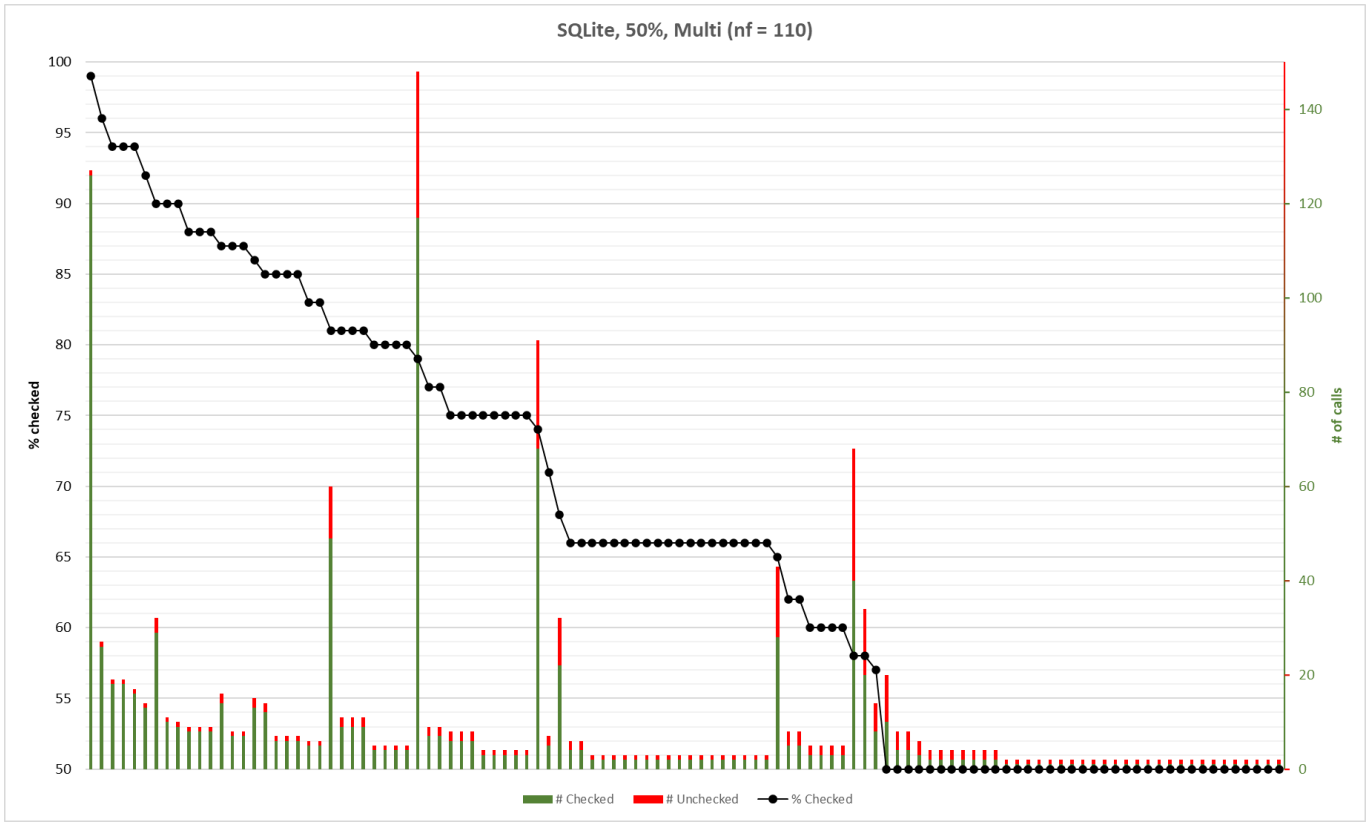

QtBase, 50%, Single (nf = 787)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

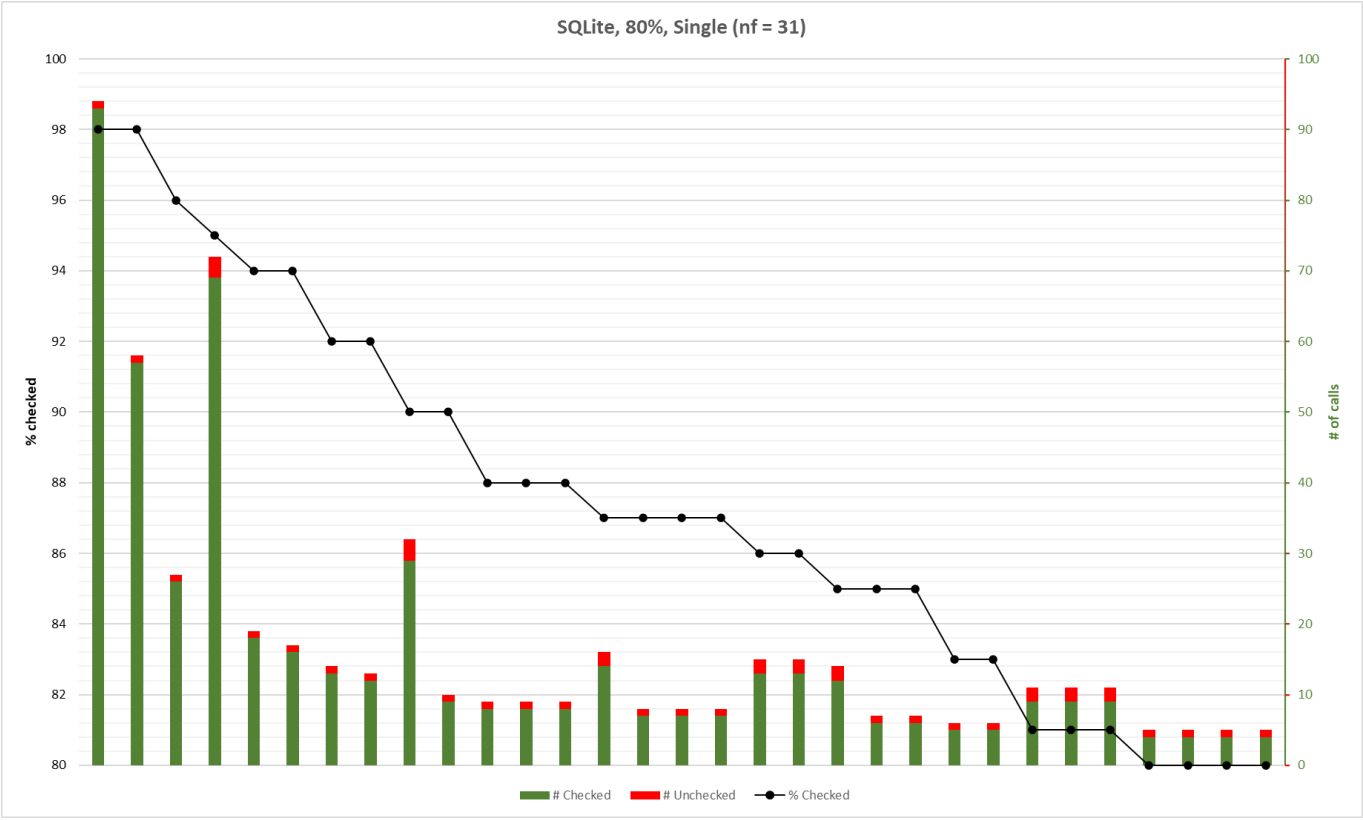

QtBase, 50%, Multi (nf = 496)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
35000
30000
25000
20000
15000
10000
5000
0
Checked
Unchecked
% Checked

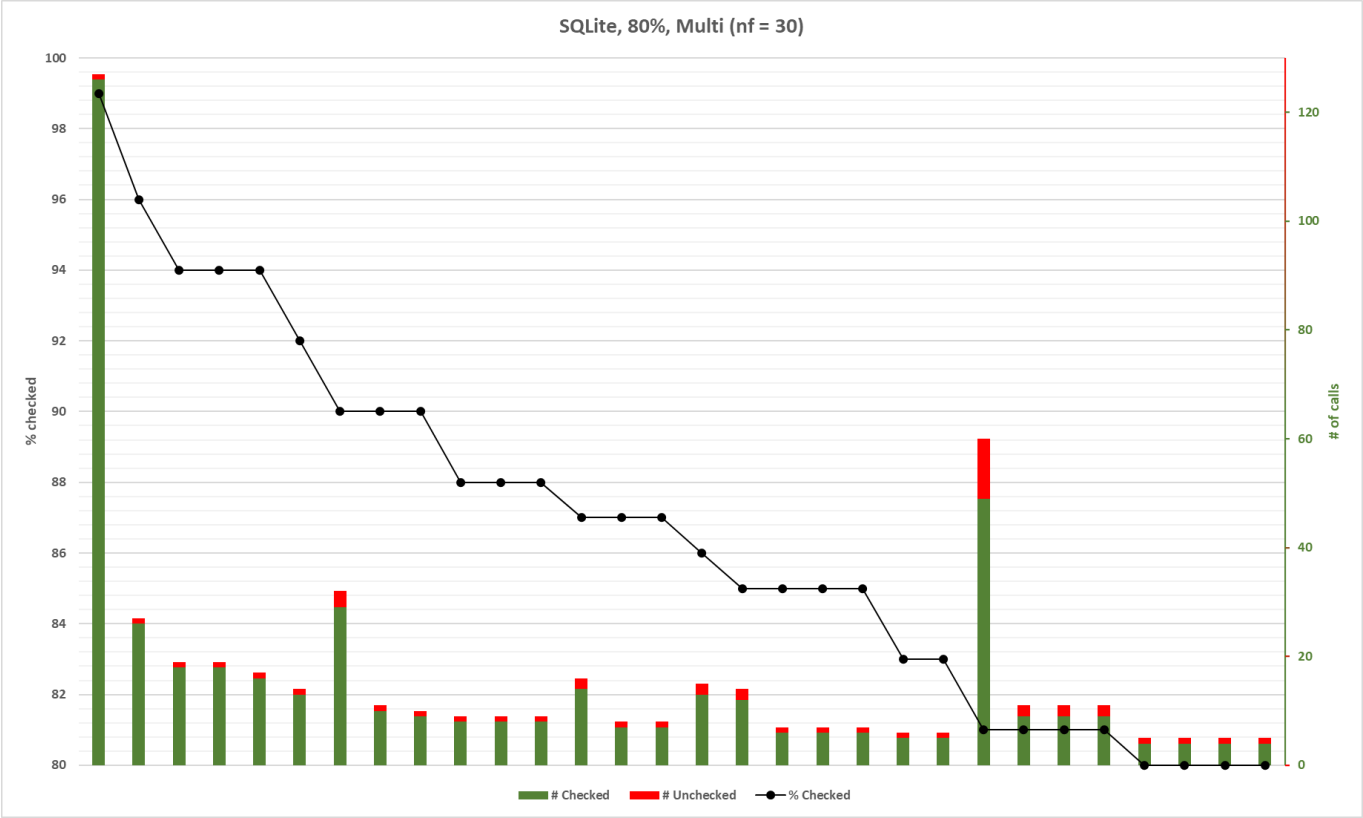

QtBase, 80%, Single (nf = 141)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

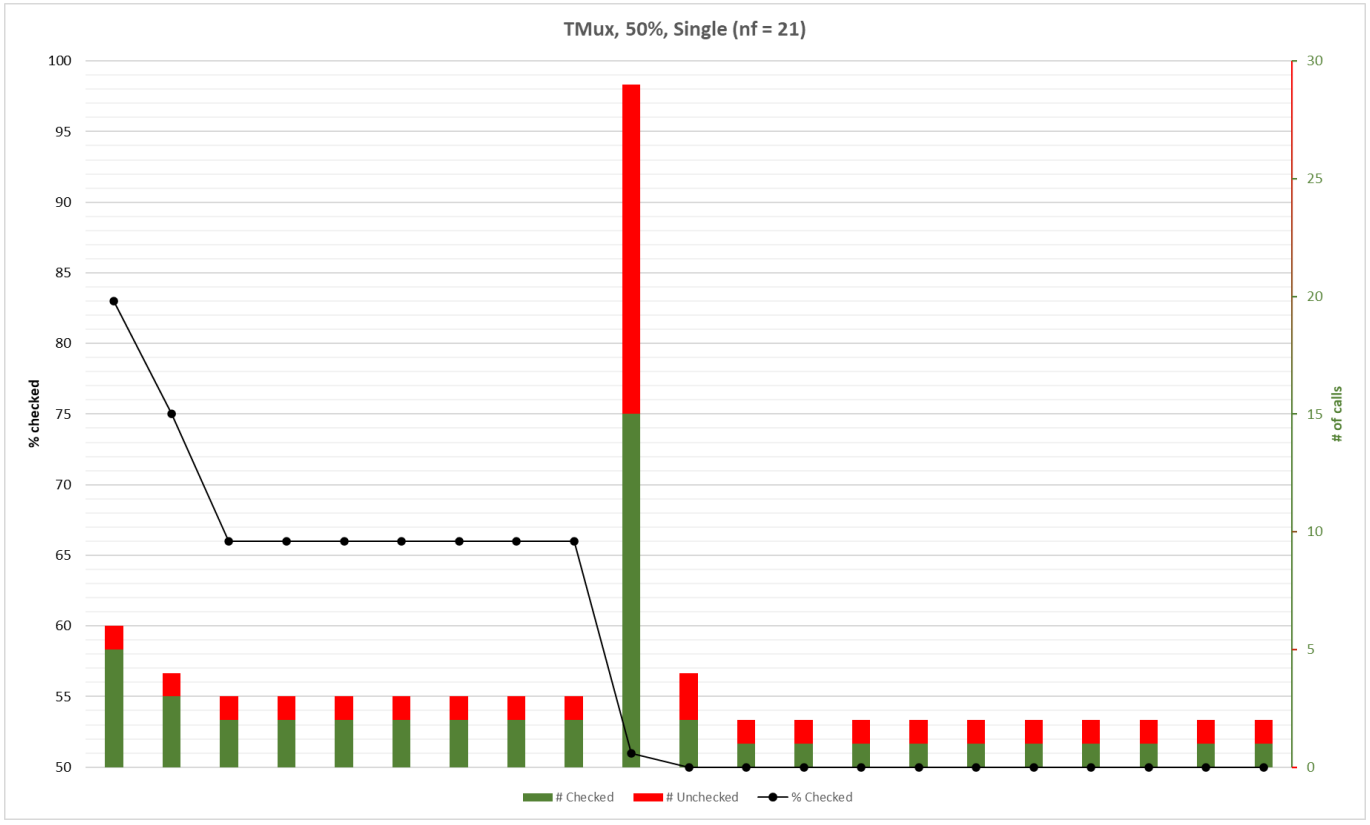

QtBase, 80%, Multi (nf = 122)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Checked
Unchecked
% Checked

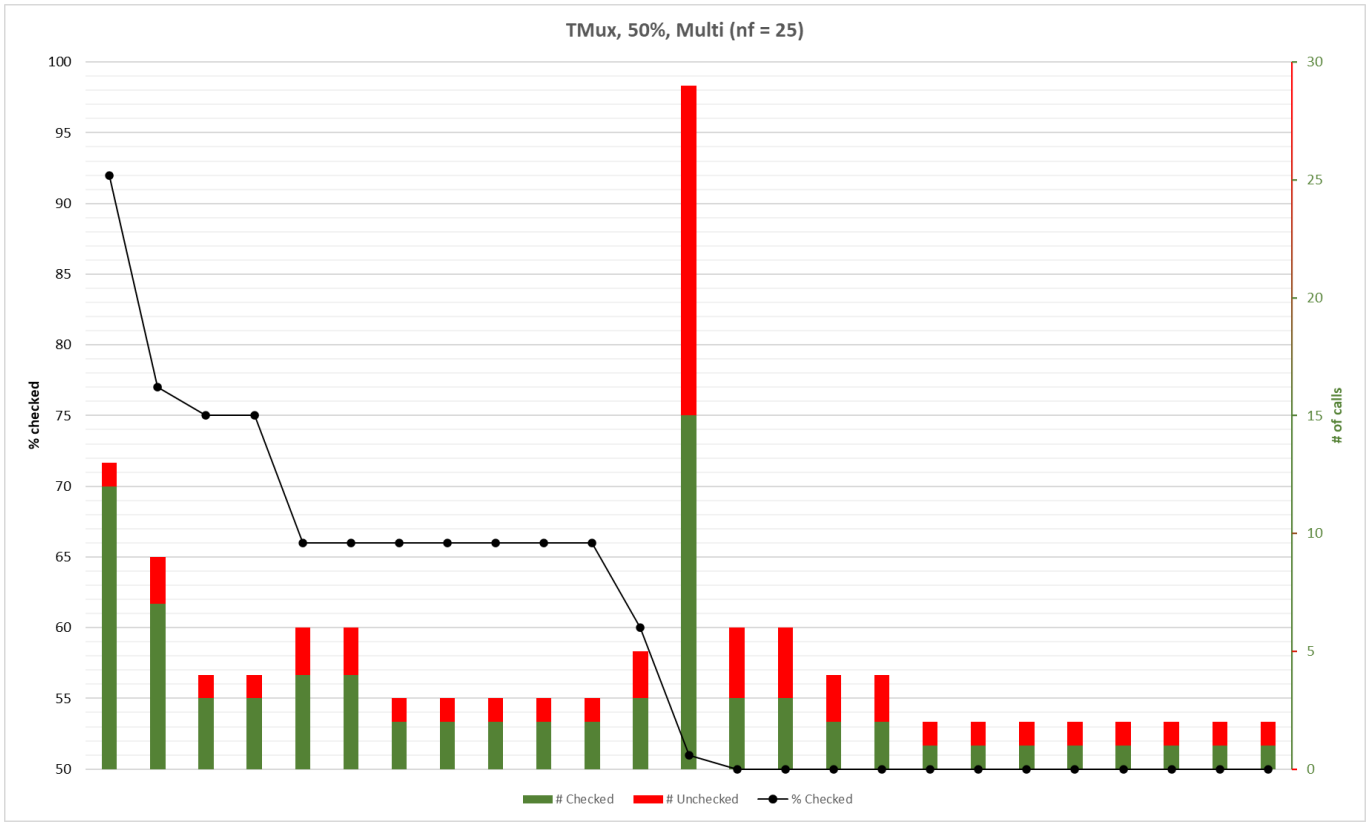

SQLite, 50%, Single (nf = 121)
% checked
# of calls
# Checked
# Unchecked
% Checked

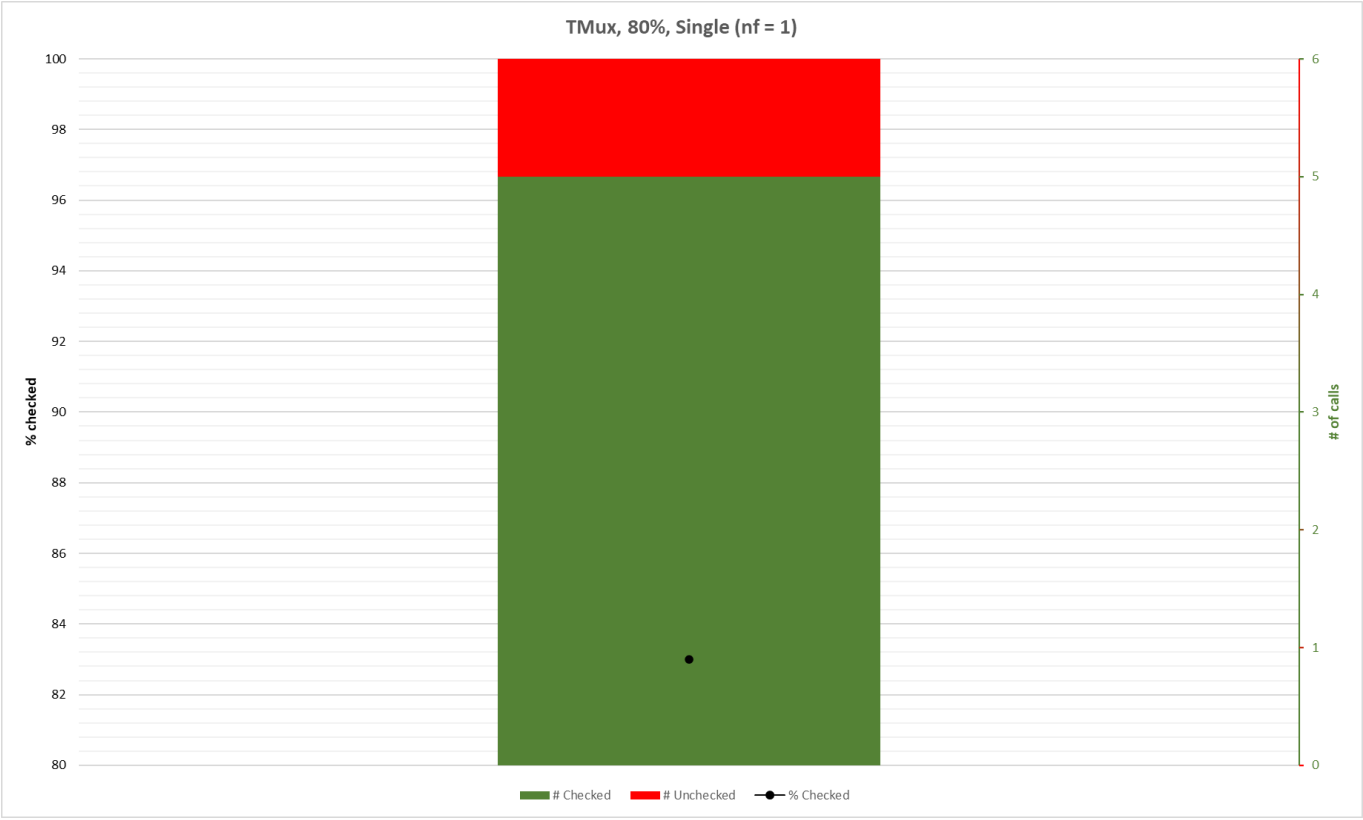

SQLite, 50%, Multi (nf = 110)
% checked
# of calls
# Checked
# Unchecked
% Checked

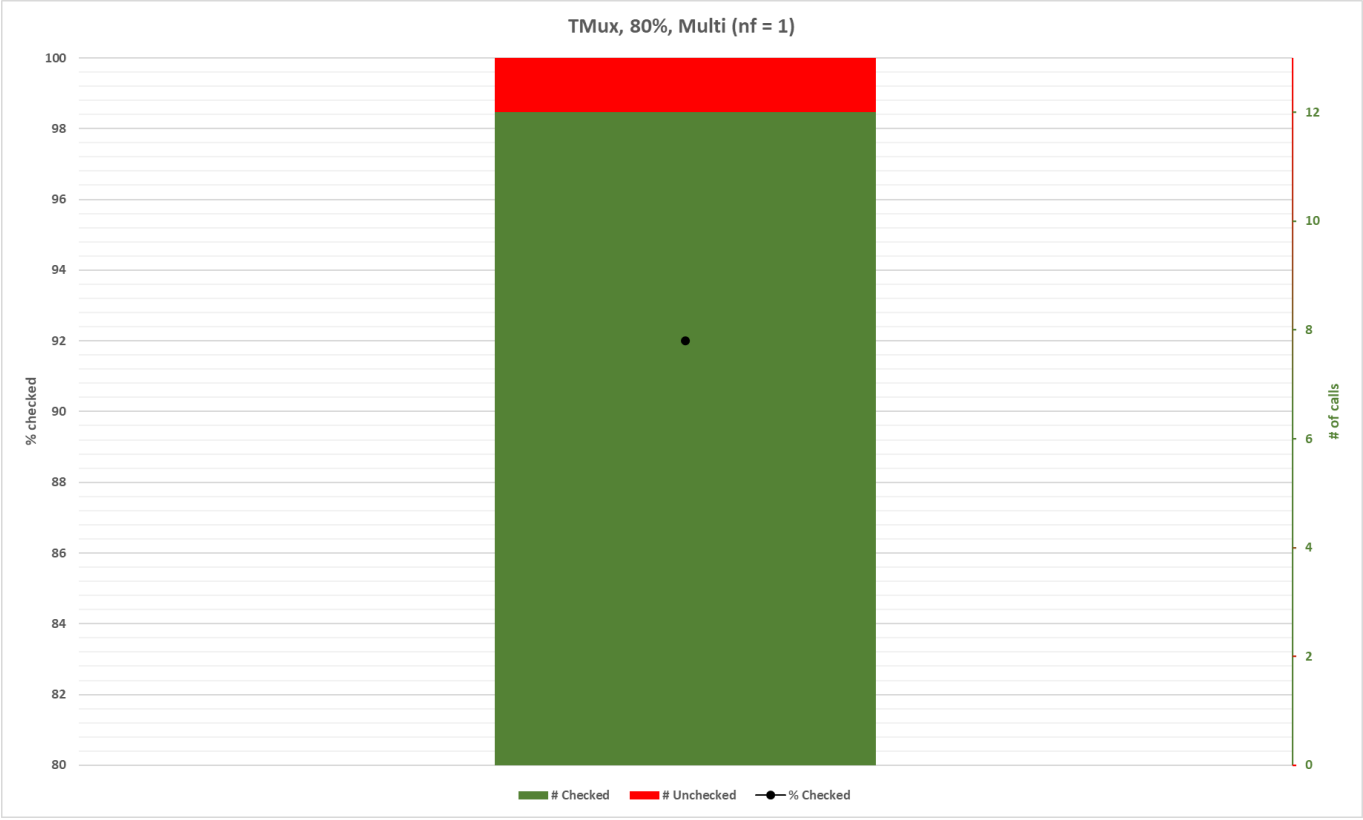

SQLite, 80%, Single (nf = 31)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
100
90
80
70
60
50
40
30
20
10
0
Checked
Unchecked
% Checked

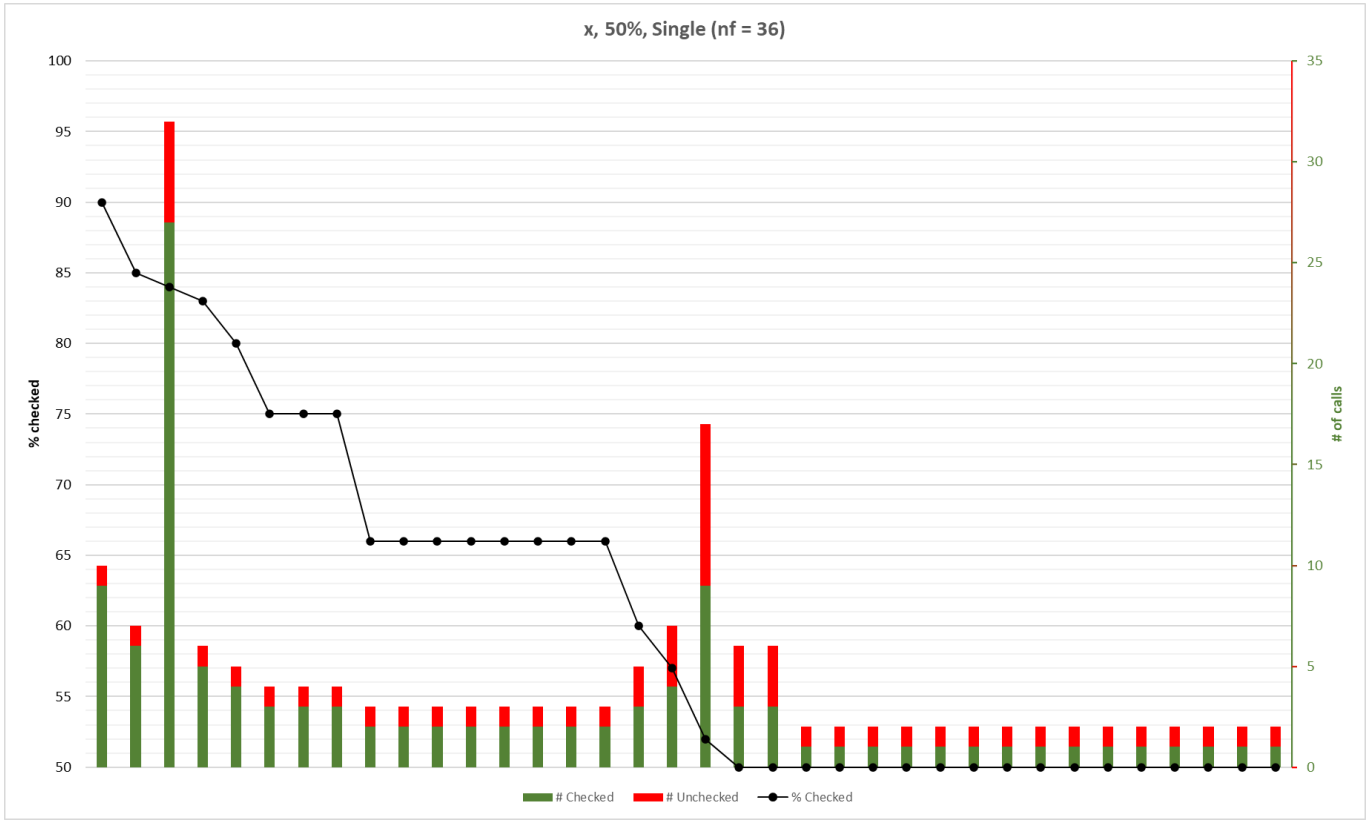

SQLite, 80%, Multi (nf = 30)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

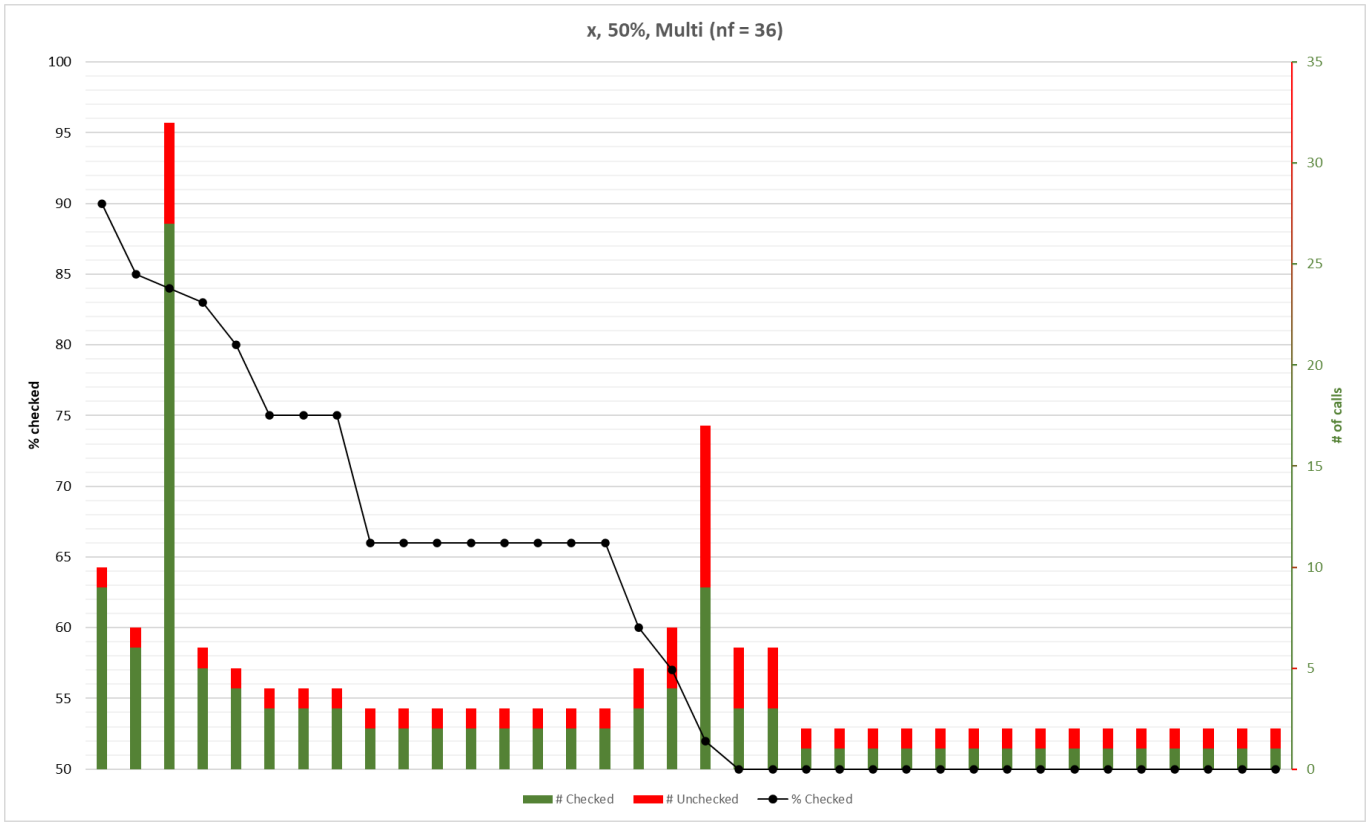

TMux, 50%, Single (nf = 21)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
30
25
20
15
10
5
0
Checked
Unchecked
% Checked

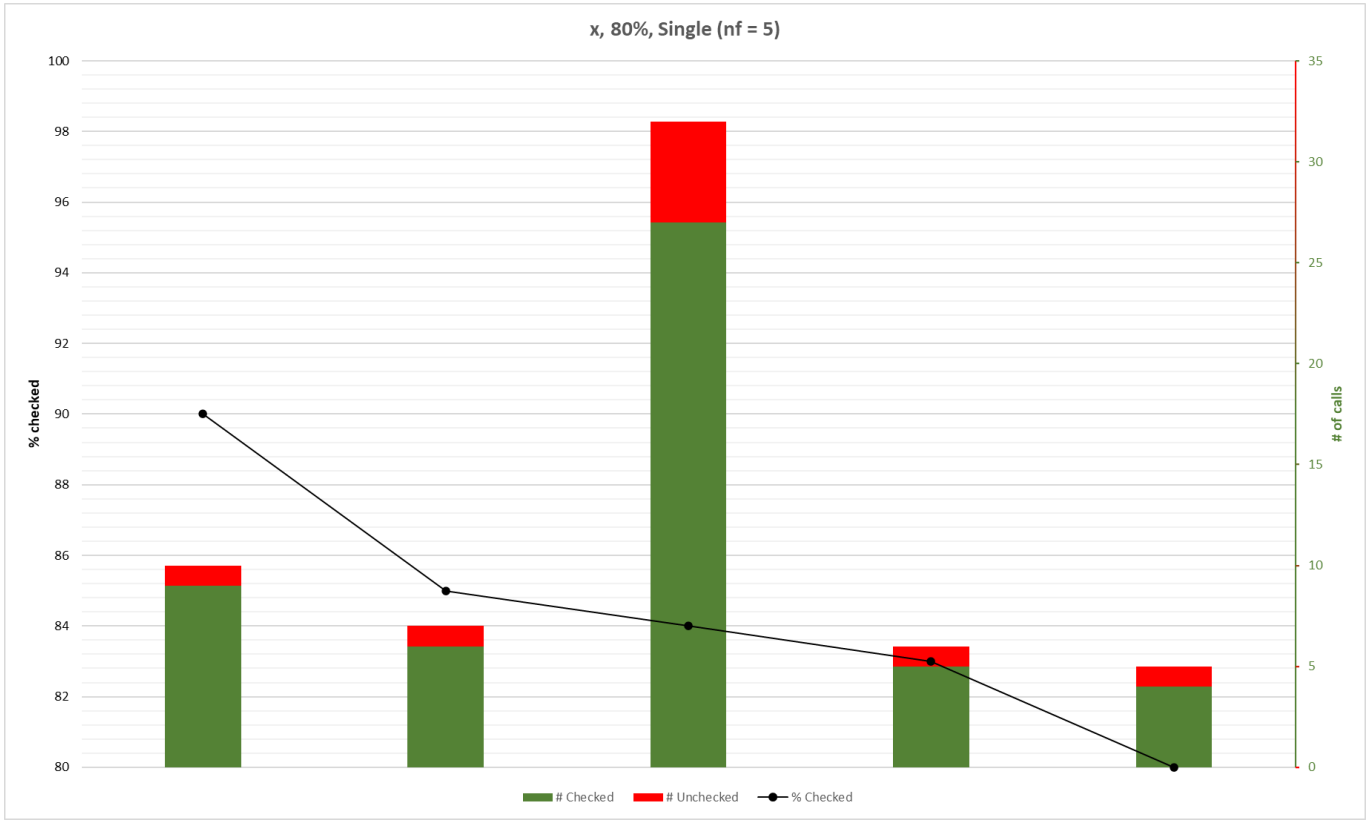

TMux, 50%, Multi (nf = 25)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
30
25
20
15
10
5
0
Checked
Unchecked
% Checked

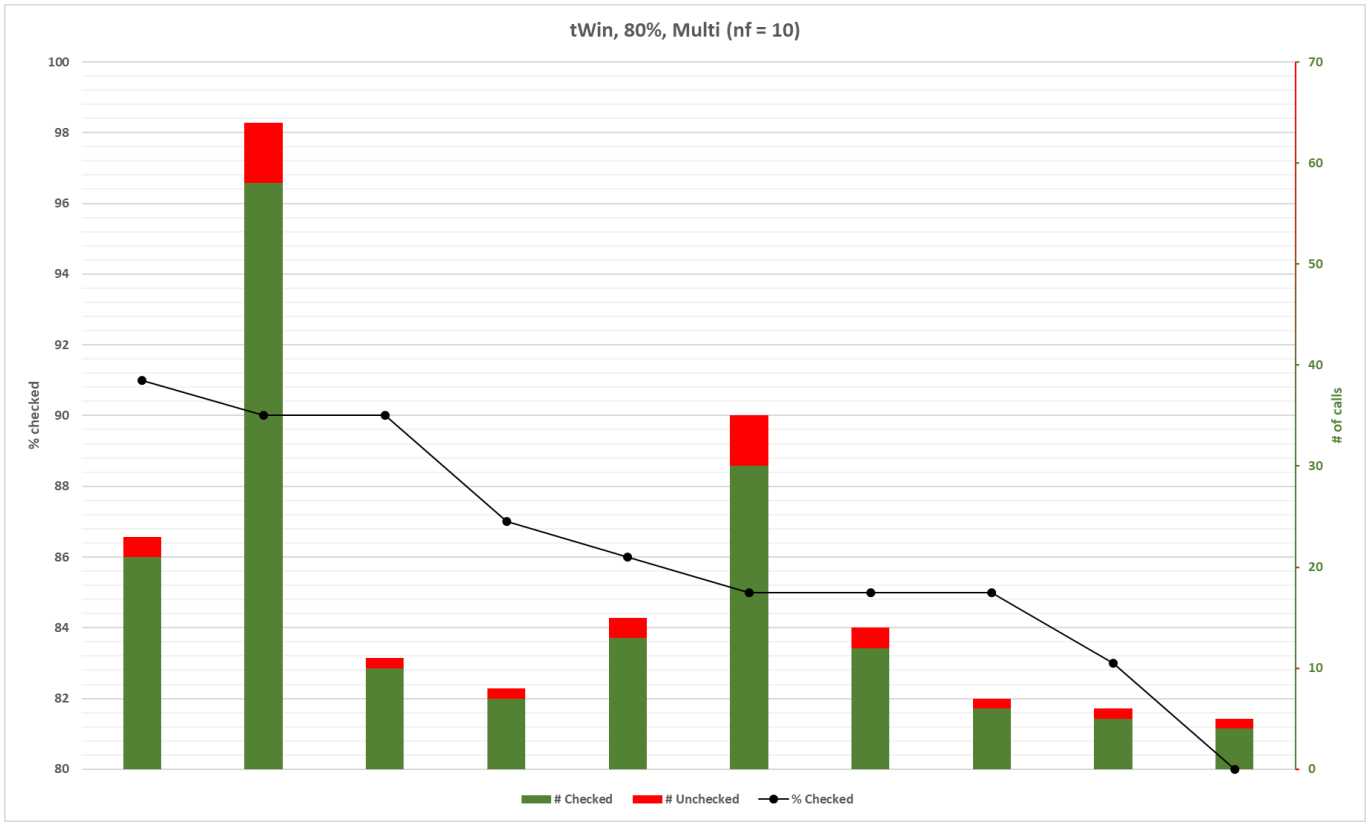

TMux, 80%, Single (nf = 1)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
6
5
4
3
2
1
0
Checked
Unchecked
% Checked

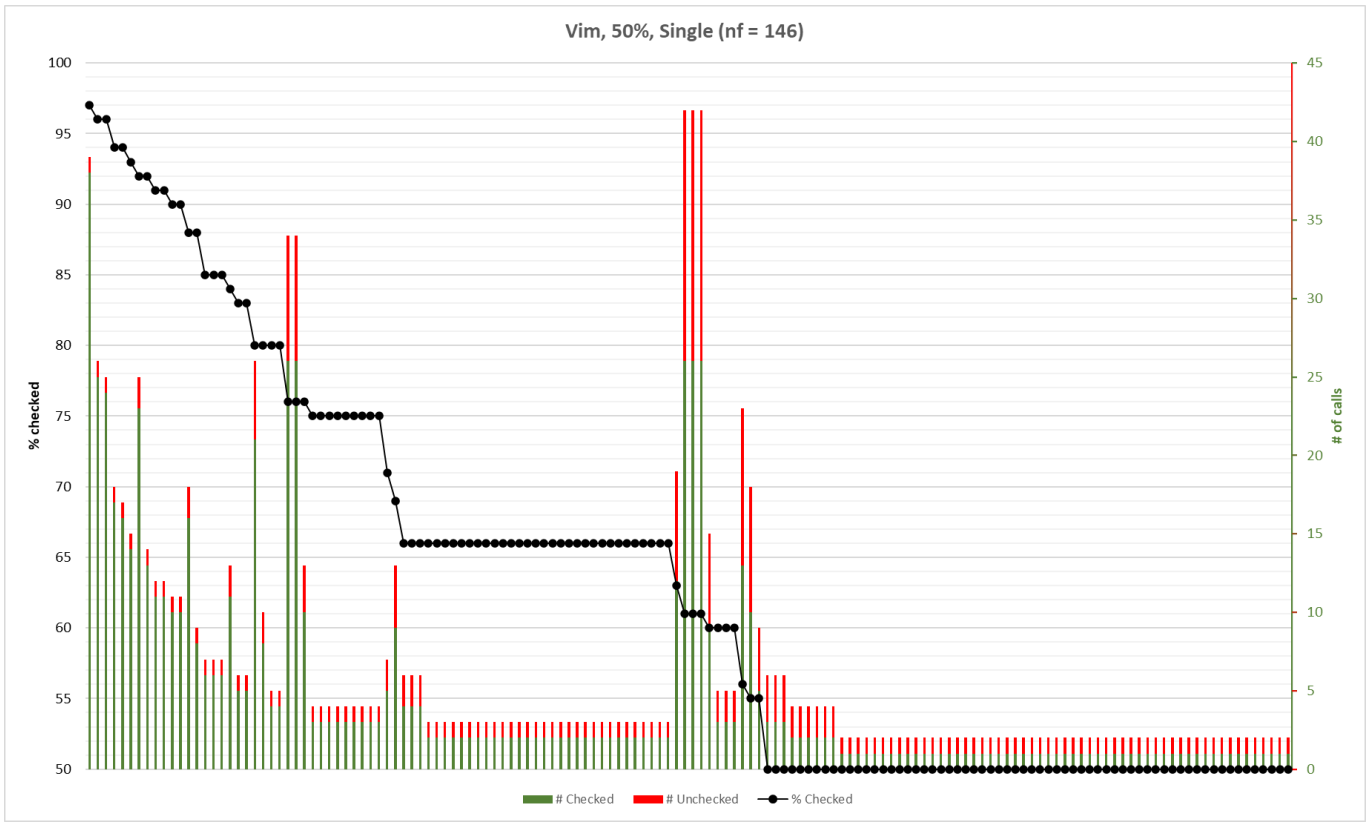

TMux, 80%, Multi (nf = 1)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
12
10
8
6
4
2
0
Checked
Unchecked
% Checked

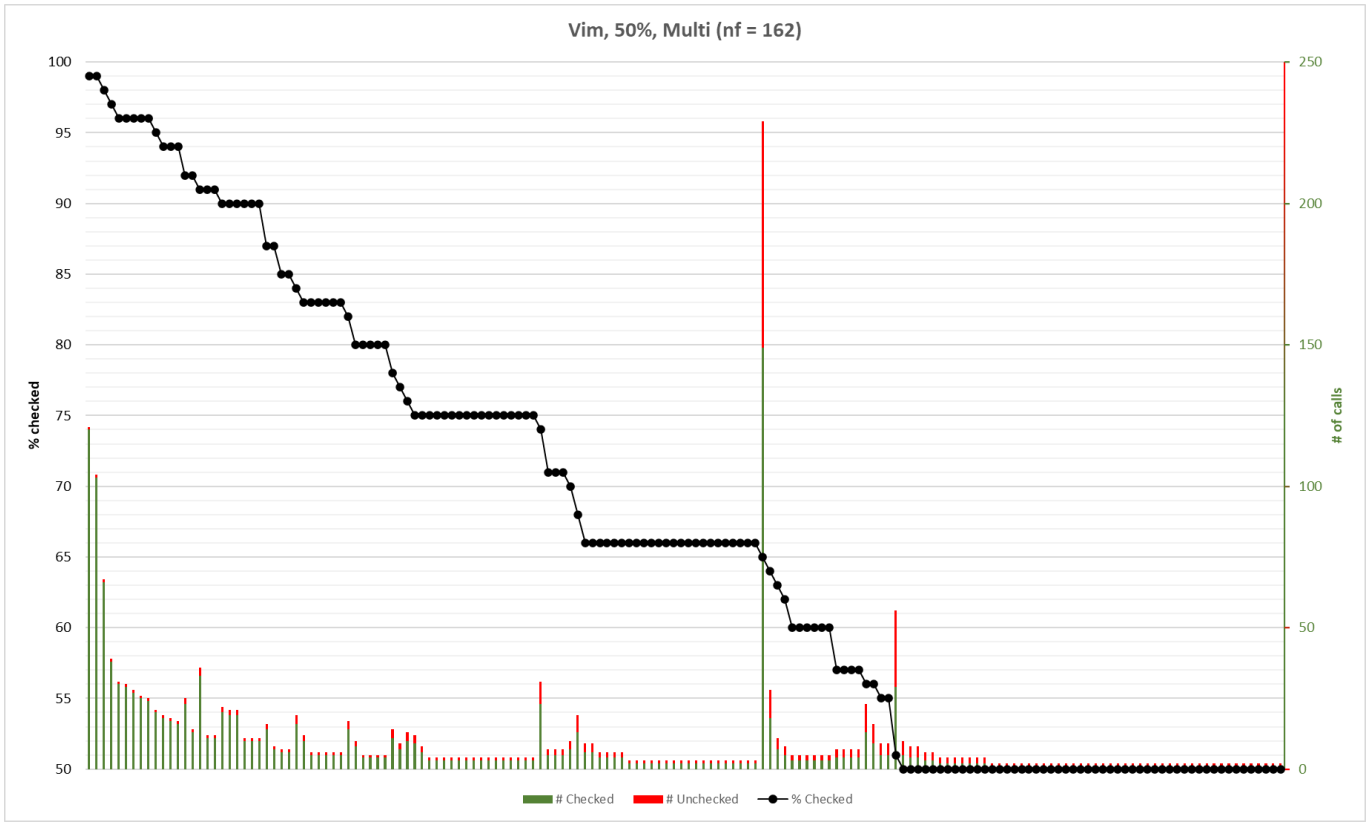

x, 50%, Single (nf = 36)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked

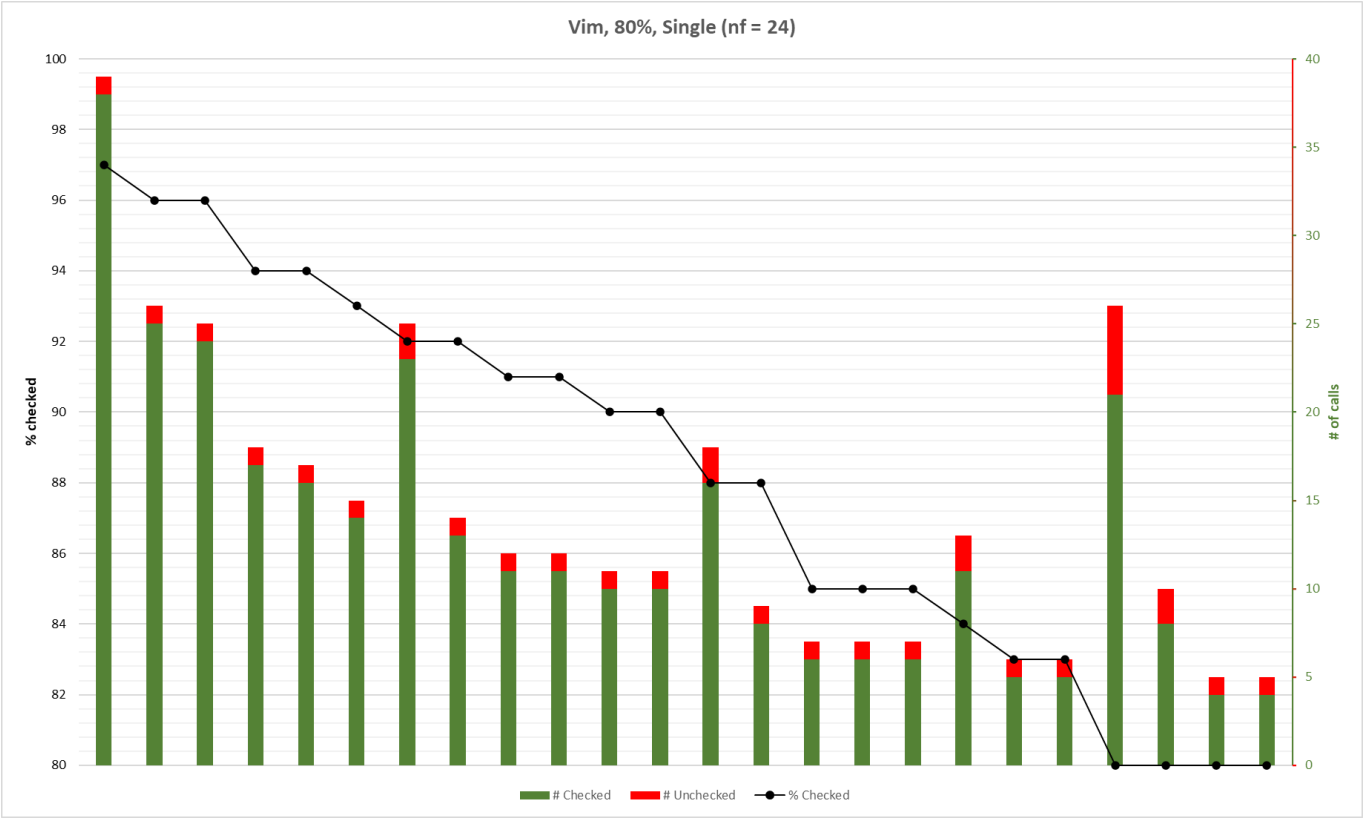

x, 50%, Multi (nf = 36)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked

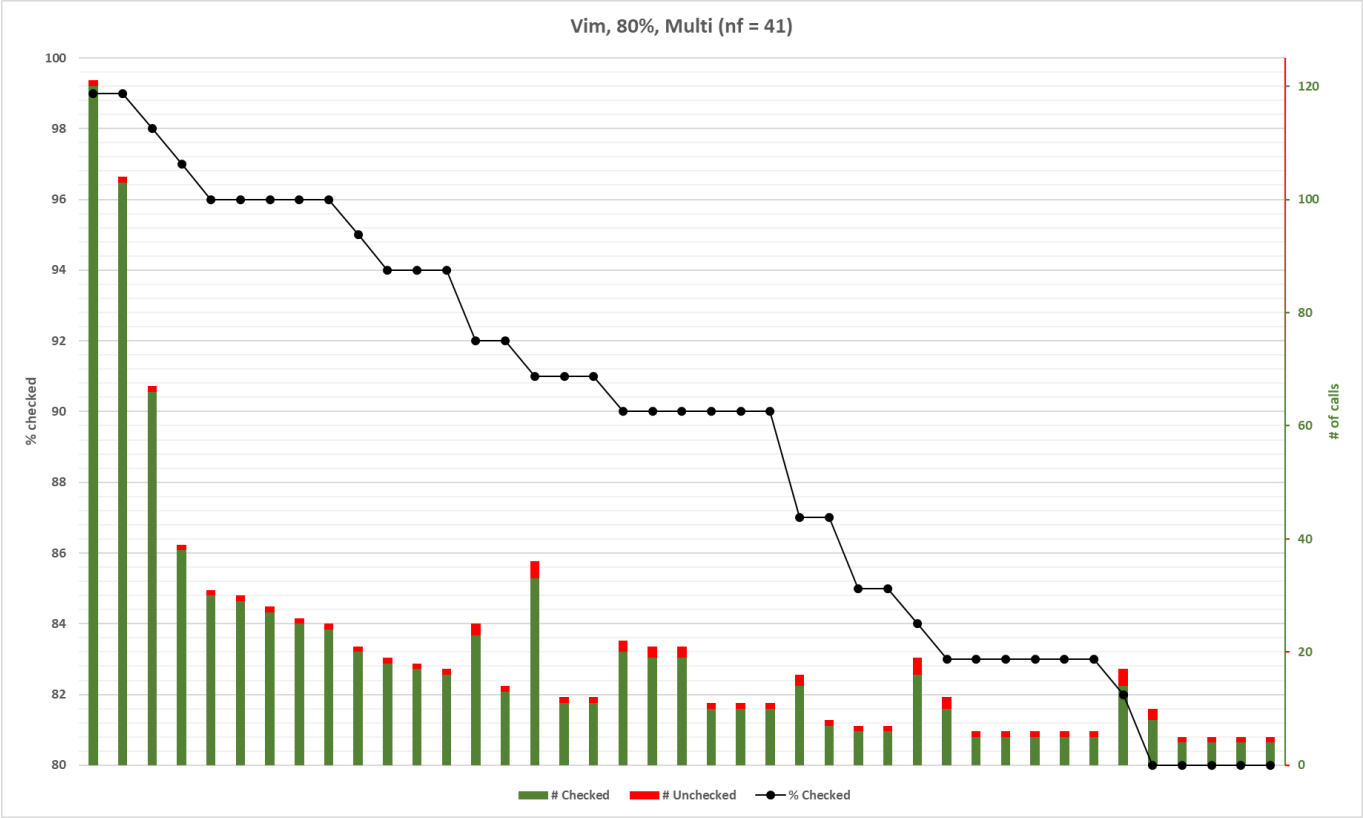

x, 80%, Single (nf = 5)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked

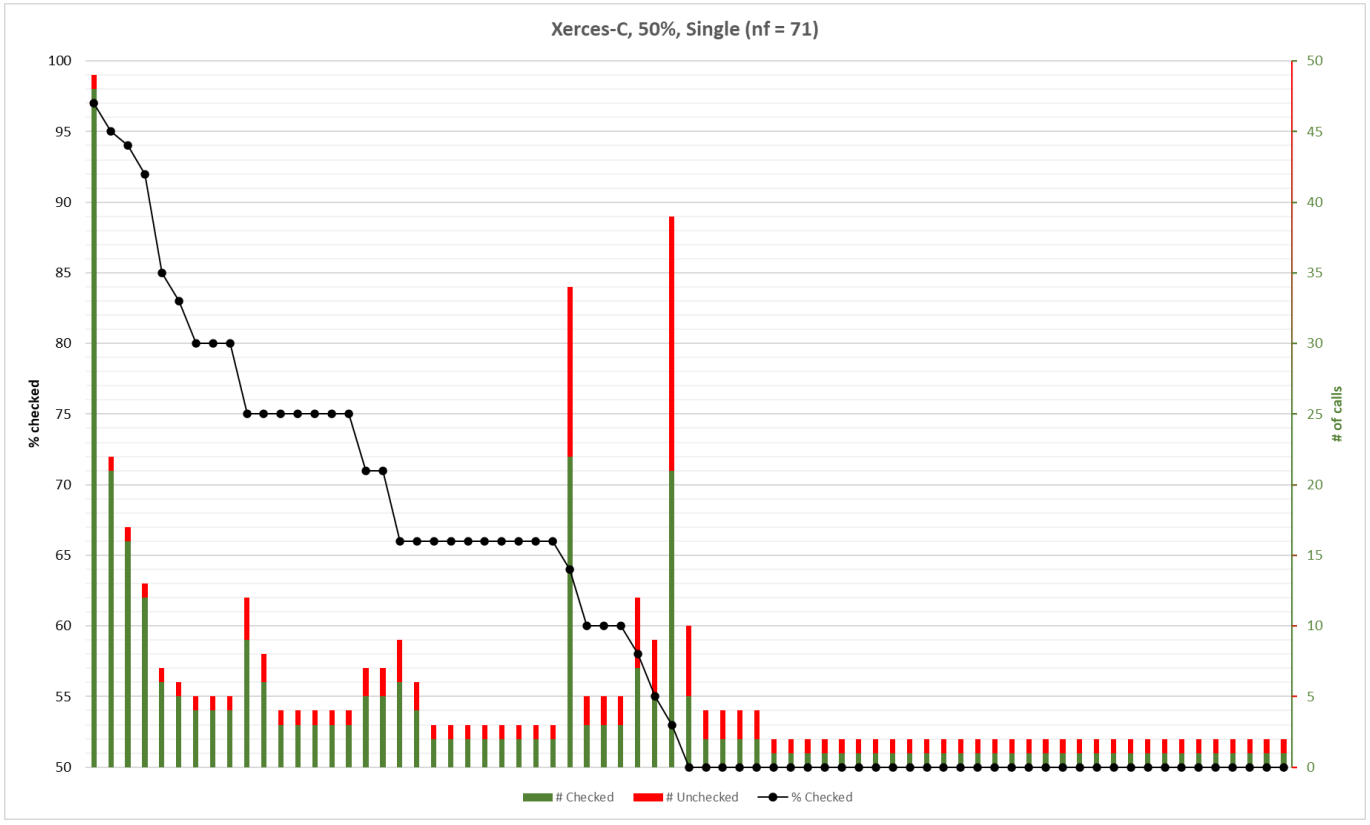

tWin, 80%, Multi (nf = 10)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
70
60
50
40
30
20
10
0
Checked
Unchecked
% Checked


Vim, 50%, Single (nf = 146)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked

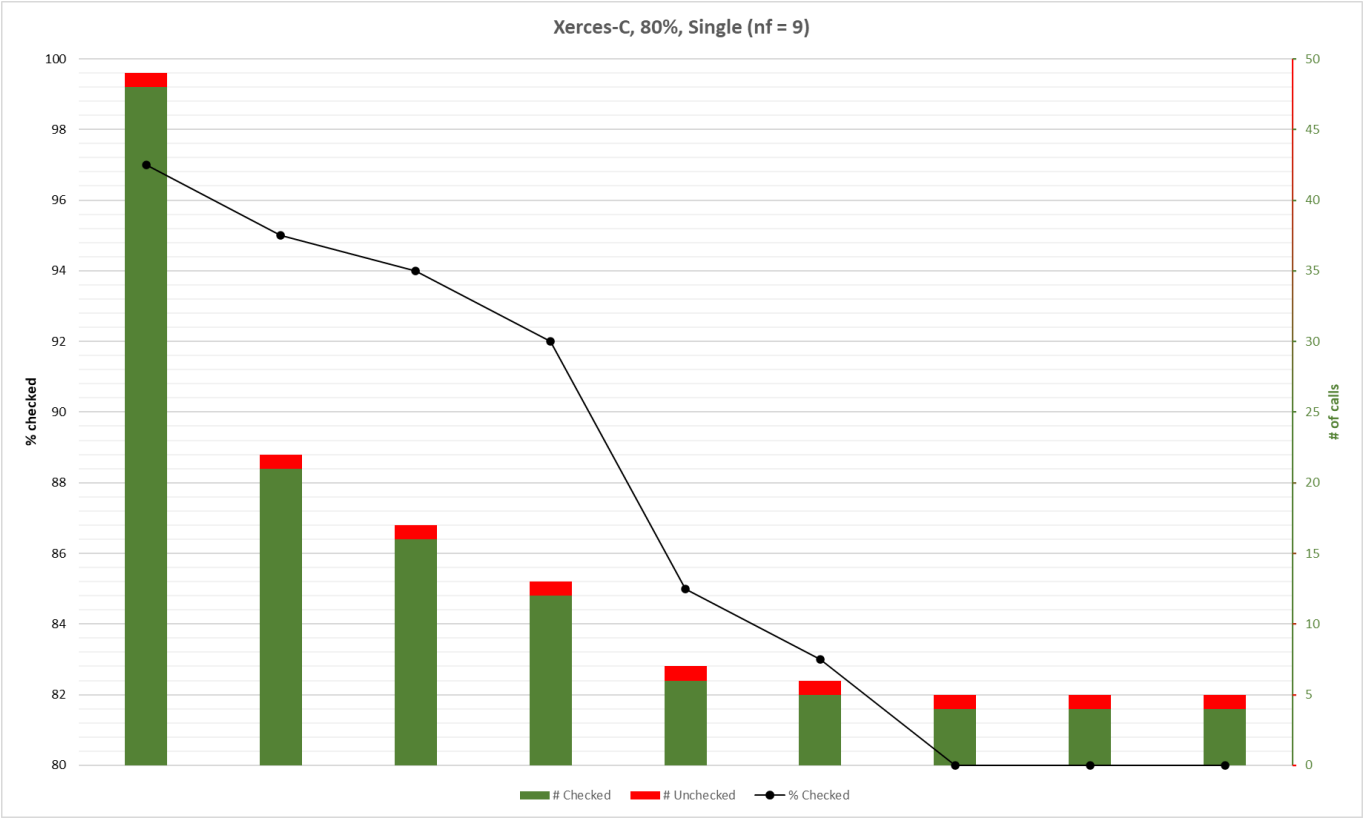

Vim, 50%, Multi (nf = 162)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
250
200
150
100
50
0
Checked
Unchecked
% Checked

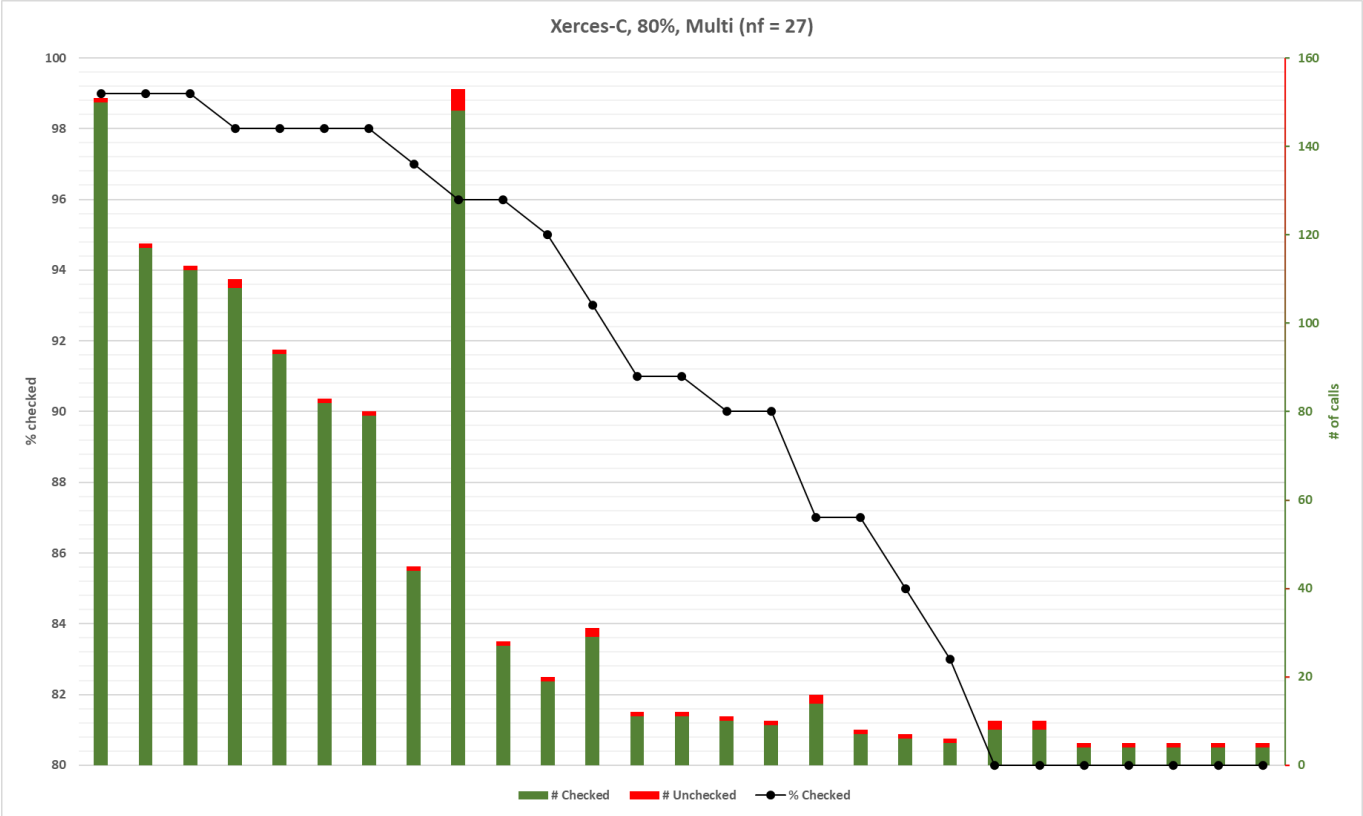

Vim, 80%, Single (nf = 24)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
40
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked


Vim, 80%, Multi (nf = 41)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
120
100
80
60
40
20
0
Checked
Unchecked
% Checked


Xerces-C, 50%, Single (nf = 71)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
50
45
40
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked


Xerces-C, 50%, Multi (nf = 63)
% checked
# of calls
100
95
90
85
80
75
70
65
60
55
50
160
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked


Xerces-C, 80%, Single (nf = 9)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
50
45
40
35
30
25
20
15
10
5
0
Checked
Unchecked
% Checked


Xerces-C, 80%, Multi (nf = 27)
% checked
# of calls
100
98
96
94
92
90
88
86
84
82
80
160
140
120
100
80
60
40
20
0
Checked
Unchecked
% Checked

Bibliography

[1] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[2] *C++ attribute: nodiscard (since C++17)*. URL: https://en.cppreference.com/w/cpp/language/attributes/nodiscard.

[3] Valentina Lenarduzzi et al. "A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision". In: (2021). URL: https://doi.org/10.48550/arXiv.2101.08832.

[4] Andrew Rice et al. "Detecting Argument Selection Defects". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017). DOI: 10.1145/3133928. URL: https://doi.org/10.1145/3133928.

[5] Hyrum Wright. "Incremental Type Migration Using Type Algebra". In: 2020.

[6] Hyrum Wright et al. "Large-Scale Automated Refactoring Using ClangMR". In: *Proceedings of the 29th International Conference on Software Maintenance*. 2013.

[7] Andrew Rice et al. "Detecting Argument Selection Defects". In: *Proc. ACM Program. Lang.* 1.OOPSLA (2017). DOI: 10.1145/3133928. URL: https://doi.org/10.1145/3133928.

[8] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150.

[9] Benedek Attila Bahrami. *[clang-tidy] Add infrastructure support for running on project-level information*. Revision on Phabricator for review. 2022. URL: https://reviews.llvm.org/D124447.

[10] Benedek Attila Bahrami. *[clang-tidy] Add the misc-discarded-return-value check*. Revision on Phabricator for review. 2022. URL: https://reviews.llvm.org/D124446.

[11] Benedek Attila Bahrami. *[clang-tidy] Add project-level analysis support to misc-discarded-return-value*. Revision on Phabricator for review. 2022. URL: https://reviews.llvm.org/D124448.

[12] *Cross Translation Unit (CTU) Analysis*. URL: https://clang.llvm.org/docs/analyzer/user-docs/CrossTranslationUnit.html.

[13] *Coverity Scan - Static Analysis*. URL: https://scan.coverity.com/o/oss_success_stories/69.

[14] *codechecker*. URL: https://github.com/Ericsson/codechecker.git.

[15] *AST Matcher Reference*. URL: https://clang.llvm.org/docs/LibASTMatchersReference.html.

[16] *bitcoin*. URL: https://github.com/bitcoin/bitcoin.git.

[17] *contour*. URL: https://github.com/contour-terminal/contour.git.

[18] *curl*. URL: https://github.com/curl/curl.git.

[19] *ffmpeg*. URL: https://github.com/FFmpeg/FFmpeg.git.

[20] *libwebm*. URL: https://github.com/webmproject/libwebm.git.

[21] *llvm-project*. URL: https://github.com/llvm/llvm-project.git.

[22] *memcached*. URL: https://github.com/memcached/memcached.git.

[23] *mongo*. URL: https://github.com/mongodb/mongo.git.

[24] *openssl*. URL: https://github.com/openssl/openssl.git.

[25] *postgres*. URL: https://github.com/postgres/postgres.git.

[26] *protobuf*. URL: https://github.com/steakhal/protobuf.git.

[27] *qtbase*. URL: https://github.com/qt/qtbase.git.

[28] *sqlite*. URL: https://github.com/sqlite/sqlite.git.

[29] *tmux*. URL: https://github.com/tmux/tmux.git.

[30] *twin*. URL: https://github.com/cosmos72/twin.git.

[31] *vim*. URL: https://github.com/vim/vim.git.

[32] *xerces*. URL: `https://github.com/apache/xerces-c.git`.

List of Figures

List of Tables

List of Algorithms

List of Codes