Report for:

Huawei MA5800 Code Evaluation & Build Engineering Assessment

Huawei Technologies

October 2019

Version: 3.0

Executive Summary

This report presents the findings of the Huawei MA5800 Code Evaluation & Build Engineering Assessment conducted on behalf of Huawei Technologies. The assessment was conducted between 15/07/2019 and 09/08/2019 (three weeks elapsed) and was authorised by Huawei Technologies.

The analysis performed regarded the code base, build process repeatability and binary reproducibility of firmwares for the MA5800 OLT (Optical Line Terminal) device. The physical device itself was not reviewed as part of this work.

The assessment consisted of an evaluation of various factors for each code base according to criteria and indexes which were chosen by Huawei. The indexes and the analyses performed against them are explained in the relevant areas of the Technical Details, Section 2. A range of data relating to code maintainability indexes was collected from the source code for R19 of the MA5800 code base, analysed and compared against the previous release R17, where applicable. This analysis of the code bases reflects a particular point in time as defined by the versions supplied for review by Huawei and does not represent a conclusive review of all maintainability aspects.

A review was made of the use of safe and unsafe C programming language functions in the source code, and assessed with respect to Huawei's own secure coding goals and their relative frequency of use in relation to the previous code base.

Additional analysis was undertaken to review the manual build process for R19, including a review of build reproducibility. This verified whether it was possible to reproduce equivalent binaries by accurately following the provided steps and checked the level of privileges required to reproduce the build. The analysis also reviewed the use of secure compilation options, the security of the build environment and whether versions of open source software in use within the device were affected by known security issues. This analysis was performed against a documented build process provided by Huawei.

Assessment Summary

Note the Caveats, Section 1.2.

Code Maintainability Indexes

Non-Blank Non-Comment Lines Per Function

For Product and Platform, the rate of NBNC lines per function was 28 lines for Product, and 29 lines for Platform. R19 successfully maintained or improved on the rates found in R17.

Reductions in volume of NBNC were identified in R19 in the Platform code bundle, including a significant reduction in NBNC line count per function between the GMDB module (from R17) and DCDM (which replaced GMDB in R19), as well as for DOPRA. This overall contributed to a total reduction in the average NBNC function line count for Platform between R17 and R19 of 17% in Platform and 1.23% in Product. This NBNC metric represents a limit on the length of a given function in real terms (ignoring white spaces).

Cyclomatic Complexity

For each of Product and Platform, a Cyclomatic Complexity Score was calculated for each function identified. The score for all functions was around 5, demonstrating that R19 maintained the same standard of Cyclomatic Complexity as the previous R17 release. It is noted that the Visual Studio code

maintainability warning (CA1502) reports a violation when the Cyclomatic Complexity is more than 25.[1]

File and Code Duplication

A reduction in duplication was observed within the R19 source code, both in terms of file duplication and code duplication.

Safe and Unsafe Functions

The R19 code base showed substantial reductions in the presence of safe and unsafe functions, almost eliminating instances of unsafe function use, while safe function use increased substantially compared to R17. The extent of the changes between the code bases (and in the number of Lines of Code) precludes simple comparisons. It is not possible to produce a single figure indicating the change in the use of safe and unsafe functions across the different code bases (and the different components of those code bases). These changes are discussed further in Section 2.2.1 where the greater detail of the discussion is better suited to drawing fair comparisons.

Static Analysis Tool Output

A reduction in the warnings issued by static analysis was observed in R19. The internal Huawei CodeMars tool was used to analyse the R17 and R19 code bases for both Product and Platform. A large decrease in the number of warnings and errors reported by the tool was noted in R19 compared to R17. Once again, given the extent of the changes between the code bases, the differences between the code bases are most accurately presented in the more detailed discussion and tables provided in Section 2.2.2.

Open Source Software

There was only one duplicate open source or third party package across R19. Additionally, the effort to reorganise the source code into identifiable directories meant that it was feasible in R19 to deterministically audit the status of dependencies in use.

Security Related Compilation Options

Overall, R19 has increased the use of security compilation options in both the Product and Platform components compared to R17. The comparison analysis made for R17 and R19 confirmed that Huawei had improved the security measures for all binaries which were part of the Platform and the Product code base.

On the Platform side, the use of security options had increased. For example, within Platform, Stack Canary use was at 1.68% in R17 RTOS, but was at 83.14% in R19 RTOS. Similarly, Stack Canary use was at 0% in R17 VRP, but was at 91.54% in R19 VRP. R17 RTO had a FULLRELRO ratio of 1.32% versus 99.60% in R19.

On Product, security options were used in a large percentage of binaries and there was only a small margin left for improvement in R19. For example, in Product, R19 MPLX, the FULLRELRO ratio was at 89.32% with Stack Canary use at 82.50%. In Product R19 MPSC, the figures were FULLRELRO at 89.27% and Stack Canary use at 82.42%.

The use of the NX flag, which prevents code execution from the stack and heap, was 100% in both R17 and R19, this mitigation makes exploitation of buffer overflows more challenging.

However, improvement can be made in the use of PIE/PIC options – the Product code usage was 98.64%. Certain modules, notably RTOS, only had a 50.16% coverage of this important option.

[1] https://docs.microsoft.com/en-gb/visualstudio/code-quality/ca1502-avoid-excessive-complexity

Discussions with Huawei indicated that these figures may have arisen because of a limitation in the version of the Checksec tool used for the assessment. However, no definitive statement can be made as to whether these figures were simply an artefact of the tool, or did reflect some underlying issue.

A more detailed discussion of these improvements is provided in Section 2.2.4 with tables breaking the improvements down by the code base components.

Build Process Engineering

The build process was checked to determine if the software could be built without the need to elevate privileges. This is because from a technical perspective there is no reason why the build process should need to done by a privileged user.

It was discovered that Huawei's build process, as documented, required the use of elevated user privileges to set up the build environment for both the Platform and Product code bases. This was discussed with Huawei and they were aware of the issue.

A proposed solution was presented by Huawei along with a working example. This solution would remove the requirement to use a privileged account for environment setup. Consequently, the provisions within the environment to facilitate the use of a privileged account could be eliminated entirely. This would in turn mean that the build process could be performed entirely by an unprivileged user. However, this solution was not implemented in R19 at the time this assessment was conducted.

Reproducible Builds

With the current setup, following the instructions and with the correct parameters, firmware binaries were successfully reproduced.

It was possible to create reproducible and identical builds as verified by the SHA-256 hash. It was required that users use a preconfigured virtual machine and follow all compilation instructions, including compiling in the correct directory. In Product, the use of a specific time stamp was required in order to replicate the build fully.

Handling of Compiler Warnings

The Platform build has improved its already clean build log output by building with no warnings. This was noted as an improvement in R19 over R17.

The number of warnings produced in Product was also reduced in R19. However, it should be noted that some of this was a result of warnings being selectively and globally disabled during the build process. It is acknowledged that it can be useful to supress some warnings to ensure that those build messages considered most important are given due prominence. However, it is important that any decision to remove a flag or suppress a warning is done selectively and that this is monitored and reviewed. It should always be expected that warnings must be individually evaluated on a case by case basis.

The examination of the R19 Platform compile process also identified areas in which GCC's features could be used to generate more useful warnings regarding the use of secure C library functions. However, it is acknowledged that this could come at the cost of an increased false positive rate.

TABLE OF CONTENTS

Document Control

Issue No.	Issue Date	Change Description
1.0	14/08/2019	Original report released to client
2.0	02/10/2019	Updated report released to client
1.0	02/10/2019	Distributable report copy released as v1.0
2.0	12/11/2019	Distributable report copy updated and released as v2.0
3.0	26/11/2019	Distributable report copy updated and released as v3.0

1 Technical Summary

The information assurance vendor was contracted by Huawei Technologies to conduct a security assessment of the systems within scope in order to identify security issues that could negatively affect Huawei Technologies' business or reputation if they led to the compromise or abuse of systems.

1.1 Scope

At a high level the scope for this assessment included:

- Evaluating Code Maintainability Indexes:
  - NBNC lines per function and Cyclomatic Complexity
  - NBNC Lines of Code
  - File repetition rate
  - Code repetition rate
  - Density of redundant code
  - Analysis of code maintainability indexes
- Further Specific Code Analysis:
  - Safe and unsafe functions
  - Usage of safe and unsafe functions
  - Analyse use of open source and third party software
  - Assessment of secure compilation option use
- Build Process Engineering:
  - Build process privilege requirements check
  - Review reproducible build status
  - Compiler warnings

For each area of assessment, the following requirements were also raised:

- Comparison of quantitative changes where possible between the current release (R19) and the previous release (R17)
- Qualitative assessment of R19 with respect to security best practice.

In Platform the source bundles provided for R19 were as follows:

- DCDM: DCDM_V300R001C00SPC030B600.rar
- DOPRA: DOPRASSPV300R003C30SPC210B300.zip
- RTOS: RTOS_V200R007C00SPC302B090.rar
- VRP: MA5800V100R019C10SPH102_VRP_V800_R018C17SPC865B865.rar
- VPP: VPPV300R003C28SPC205B030.zip

For Platform R17, the source bundles provided were:

- GMDB: GMDBV10R002C30SPC401B100.rar and GMDBV100R002C30SPC430B200.rar
- DOPRA: NGA17A-MA5800V100R017C10HP2057-DOPRA-20190222.rar
- RTOS: RTOS_SDK_V100R005C00SPC360.rar
- VRP: MA5800 V100R017C10SPH219_VRP_V800R013C00SPC61fB61f.rar
- VPP: VPP300R003C26SPC214B030.rar

In Product the source bundles provided for R19 were as follows:

- Product: MA5800 V100R019C10SPH102.zip
  - In Product the source bundles provided for R17 were as follows:
- Product: ATIA MA5800 V10R017C10SPH219_CSEC.zip

1.2 Caveats

It is evident that a considerable amount of work was undertaken between R17 and R19, resulting in a significant restructuring of the code base. Additionally, there was a substantial overhaul of the build process in R19. Consequently, R19 is fundamentally more assessable than R17. As a result, some results for R17 may not be accurate when used as a comparator.

Transcription Based Reporting

Due to the secure nature of the physical environment in which the assessment was performed, all results had to be transcribed manually into this report (as it was not possible to transfer information in or out electronically). Data was collected in a secure physical environment with limited access to reference materials and only pre-installed tooling. That data was then extracted for further analysis on secured PCs outside the secure test environment so the assessment team could analyse and transcribe final results into the report. This separation between reporting, analysis and source code environments introduced some risk of transcription errors and potentially of data modification.

Compiler Warnings

During the assessment, the RTOS module in Platform (R19) produced build logs but these did not include build tool command invocations and warnings. Hence, it was uncertain how reliable the results were.

Platform R17 VPP was never built successfully. Hence, no build logs could be analysed during the assessment.

Restructuring and significant changes in the build process between R17 and R19 in Product make comparisons difficult to assess.

Secure Compilation Options

Checksec analysis for comparison on the R17 Product binaries was not feasible as the unpacking process required a number of proprietary Huawei tools and a complex process which was not feasible to complete at the late testing stage when the R17 build process was finally successful.

2 Technical Details

The remainder of this document is technical in nature and provides additional detail about the items already discussed, for the purposes of remediation and risk assessment.

2.1 Evaluating Code Maintainability Indexes

2.1.1 NBNC Lines Per Function and Cyclomatic Complexity

Data collection

Data was collected from the R17 and R19 source bundles provided in both the Product and Platform secure test environments. In order to focus analysis on the code base owned by Huawei, the results were filtered where possible to eliminate those from third party dependency packages which were bundled with Huawei's own code (for example, in R19, this meant eliminating *vendor*, *open_source*, and *open3rd* folders). However, for R17, dependencies were not organised in an easily identifiable manner and may have been included accidentally in the final result.

Objective

To gather data on the density of Cyclomatic Complexity within the code base.

It should first be stated that there is no accepted consensus on whether the number of Non-Blank Non-Comment (NBNC) Lines of Code in a function has an impact on software complexity and sustainability and what the optimal length of a function should be. Nevertheless, these metrics are commonly used and in this case allowed comparisons between the code bases.

Current Status Assessment

Currently both the Platform and Product elements for R19 maintain an average Cyclomatic Complexity which is within recommended guidelines (~5 points per function, average). The average number of non-blank, non-comment lines per function for R19 stands at an average of around 28 lines for Product, and 29 lines for Platform. It is difficult to set fixed "best practice" guidelines for the number of NBNC lines per function, but literature on clean coding practices suggests a range of common recommendations between 20 and 100 lines as best practice, which this clearly falls inside of.

Improvement Analysis

For the Platform code, R19 had a significant increase in the total function count over R17, particularly with respect to considerable redevelopment of the VRP code base (which forms the bulk of the Platform code base). The average number of lines per function was reduced from 35 lines per function to 29 NBNC lines per function overall. In comparison to R17, Cyclomatic Complexity for the R19 Platform was reduced slightly, as was average NBNC lines per function.

For Product, the overall complexity retains an already optimal Cyclomatic Complexity and number of NBNC lines per function, as shown in the table overleaf.

Platform Element	Avg. Complexity	Avg. Statements/Fn	Avg NBNC Lines/Fn
Platform R17	**5.10**	**19.93**	**34.98**
GMDB	7.34	35.12	66.49
DOPRA	5.33	20.12	35.59
RTOS	5.66	94.60	104.49
VRP	5.03	19.01	33.87
VPP	4.66	13.03	26.69
Platform R19	**4.88**	**19.81**	**28.92**
DCDM	4.75	22.65	37.14
DOPRA	4.57	17.31	26.85
RTOS	4.02	14.00	19.78
VRP	4.90	20.00	29.00
VPP	5.17	16.16	26.85
Platform change	**-4.31%**	**-0.60%**	**-17%**

Element	Avg. Complexity	Avg. Statements/Fn	Avg. NBNC Lines/Fn
Product R17	4.21	16.93	28.64
Product R19	4.74	17.88	28.29
Prod. Change	**+12.59%**	**+5.61%**	**-1.23%**

2.1.2 NBNC Lines of Code

Data collection

Data for this analysis was collected on R17 and R19 source bundles provided in both the Product and Platform secure test environments. In order to focus analysis on the code base owned by Huawei, the results were filtered where possible to eliminate those from third party dependency packages which were bundled with Huawei's own code. For example, in R19, this meant eliminating *vendor*, *open_source*, and *open3rd* folders. However, for R17, dependencies were not organised in an easily identifiable manner and may have been included accidentally in the final result.

Objective

To review the average file length over the Product and Platform code bases in R19 and evaluate any relative change between the R17 and R19 code releases.

It should first be stated that there is no accepted consensus on whether the number of Non-Blank Non-Comment (NBNC) Lines of Code in a function has an impact on software complexity and sustainability and what the optimal length of a function should be. Nevertheless, these metrics are commonly used and in this case allowed comparisons between the code bases.

Current Status Assessment

The total number of lines of code was over 20 million for the whole MA5800 Platform code base, and nearly 26 million lines for the Product code base. The average number of lines of code per source file was identified as the same for both Platform and Product, at 1348 lines per file.

A small number of very long files were identified. Review of a handful of these indicated they were used to store lengthy constant tables in the source or batteries of unit tests, and consequently these did not pose a significant concern for the maintainability of the code base as they did not contribute any significant logical components to the source.

R19 Platform Element	Avg. line count	Source files	Total NBNC LoC
MA5800 (Total)	N/A	14943	11873028
DCDM	1001	164	164245
DOPRA	419	1358	570282
RTOS	235	114	26870
VPP	364	568	206925
VRP	856	12739	10904706

R19 Product	Mean line count	Source files	Total LoC
MA5800 Code Base	912	17315	15791341

Improvement Analysis

Overall between R17 and R19, the number of NBNC lines in Platform stayed around the same, at a level of less than 1000 NBNC Lines of Code for both Platform and Product.

Although there were minor changes in the average NBNC and Cyclomatic Complexity scores between R17 and R19, the average values remained within best practice, and these statistical variations are likely dominated by the extensive refactoring work that consolidated and simplified the organisation of the code base.

R17 Platform Element	Avg. line count	Source files	Total NBNC LoC
MA5800 (Total)	N/A	23145	17540640
GMDB	2892	52	150362
DOPRA	476	4335	2065296
RTOS	425	100	42506
VPP	387	785	303987
VRP	838	17873	14978489

R17 Product	Mean line count	Source files	Total NBNC LoC
MA5800 Code Base	945	37869	35795304

2.1.3 File Repetition Rate

Data collection

The source code for R17 and R19 on both Platform and Product was collected for this work item.

Objective

The objective of this item was to review the total amount of duplicated C and C++ files within the Product and Platform code base in R19 and report on the improvements or repeated issues from R17.

Repeated files create an environment where code changes must be repeated on every duplicate file in order to ensure that vulnerabilities or bugs have been fixed. If a file is missed, unexpected or dangerous behaviour could result, affecting the quality of the code base.

Current Assessment

Platform

The following table shows the total number of files, duplicate files, the percentage of duplicates, and the average and median number of file duplicates for the R19 code base.

Due to the organised structure of R19, it was possible to filter out a number of third party, open-source code which was therefore not included in the statistics shown below.

The "Mean" in the table below represents the average number of duplicate copies in a set of matching duplicates.

The "Median" in the table below represents the number of duplicate copies halfway in the list.

Project Element	C & CPP Files	Duplicated Files	Percentage	Mean	Median
DCDM	83	0	0	-	-
DOPRA	1367	21	1.54%	2.33	2
RTOS	114	0	0	-	-
VPP	568	0	0	-	-
VRP	12815	0	0	-	-

Note that, the VRP element only contained duplicate files for open-source code.

R19 had a very small amount of duplicate C and C++ files, and duplicate files were only identified in the DOPRA element. The number of duplicates in the DOPRA module was very low. Only 1.54% of the C and C++ files were found to be duplicates. This is very good practice as the chance of updating one file, and not the duplicates, was reduced.

Product

The following table shows the total number of files, duplicate files, the percentage of duplicates, and the average and median number of file duplicates for the R19 code base.

Due to the organised structure of R19, it was possible to filter out a number of third party, open-source code which was therefore not included in the statistics shown below.

Project Element	C & CPP Files	Duplicated Files	Percentage	Mean	Median
Product R19	17573	137	0.78%	3.43	2

Only a small percentage of files were found to be duplicates in Product R19.

The table below presents data for the most duplicated files in the R19 code base. The most duplicated file was an empty file. The second most duplicated file was identified after the files were filtered so as to only include those with some meaningful content. 'Meaningful content' was defined by applying arbitrary line and character criteria; it is expected that it would be beneficial to define 'meaningful content' and how empty files are treated before any future assignments.

Project Element	Max number of duplicates of single file	Line Count
Product R19 most duplicated	48	0
Product R19 most duplicated (meaningful content)	6	67

It is worth noting that, the largest number of file repetitions were for empty C files. These files could affect code metrics if not properly accounted for. Additionally, it is not normally recommended that empty C files be included in the source as they could be mishandled by developers, especially when not documented. It is recommended that the use of empty C files be reviewed, and if not required, be removed from the code base.

Improvement Analysis

Platform

The following table compares the total percentage and number of duplicate files found in R17 with those found in R19.

As can be seen below, the R19 code base had significantly less repeated files in comparison to R17. R19 had a duplicate file percentage of 0.14%, while the percentage for R17 was 19.90% duplicate files.

This is a strong improvement in R19 as less duplicate files decreases the likelihood of inconsistent changes to the code base.

Project Element	Percentage of Duplicated Files	Total number of Duplicated Files
R17	19.90%	4605
R19	0.14%	4

The change from R17 to R19 removed all duplicate files from both VPP and VRP (which had between 6% to 8% file repetitions in R17) and drastically reduced the number of repetitions from the DOPRA element, from 3061 duplicates out 4389 files to 21 file duplicates out of 1367. The statistics for R17 can be seen below.

Supplemental Data

R17 data table:

Project Element	C & CPP Files	Duplicated Files	Percentage	Mean	Median
GMDBC401B100	26	0	0	-	-
GMDBC430B200	26	0	0	-	-
DOPRA	4389	3061	69.74%	2.63	3
RTOS	176	0	0	-	-
VPP	783	50	6.39%	2	2
VRP	17746	1494	8.42%	2.05	2

Product

The following table compares the total percentage and number of duplicate files found in R17 with those found in R19.

As can be seen below, there was a significant reduction in the number of duplicate files in R19 in comparison to R17, from 34.32% of duplicates to only 0.78%. This was an improvement that increased the manageability of the code base as it is less likely that duplicate files would be updated while the others remain outdated, which is a common cause of unexpected behaviour and security flaws.

Project Element	Percentage of Duplicated Files	Total Number of Duplicated Files
R17	34.32%	13271
R19	0.78%	137

As noted above, many of the duplicate files were found to be empty C files. A reduction of empty C files was also observed in R19, which included 48 empty files instead of 395 included in R17. Although this was an improvement, it is still recommended that the use of the files and their necessity be reviewed. The statistics for R17 can be seen below.

Supplemental Data

R17 data table:

Project Element	Max number of duplicates of single file	Line Count
Product R17 most duplicated	395	0
Product R17 most duplicated (meaningful content)	43	23

2.1.4 Duplicate Code

Objective

The objective of this item was to review the rate of code repetition within the R19 Platform and Product code base and make a comparison with R17. Duplicating slices of code throughout a code base presents a number of problems in terms of optimising maintainability and security.

Identical functionality in different parts of a code base results in a lack of a single location of responsibility for handling a particular, commonly used function, increasing the potential for any code defects found in duplicated code to be present and vulnerable multiple times throughout the code base. As these defects would be spread throughout multiple files, there would be a high chance that some files would not receive a fix or update.

Data collection

The source code for R19 and R17 on both Platform and Product was collected for this work item.

Current Assessment

Product R19

The table below includes the information regarding the total number of repeated code slices (made up of 10 lines), the largest number of repetitions for a single slice, the average number of repetitions per each repeated code slice, and the median number of repetitions. These results were taken from the Product R19 source code.

Element	Repeated Slices	Maximum Slices Repetition	Mean	Median
Product R19	1306906	4402	2.63	2

The maximum number of repetitions of a slice of code found to be 4402 repetitions, the mean and median metrics provide more insight into the overall results, with the average number of repetitions nearing 3 repetitions per slice of code.

Platform R19

The table below includes the information regarding the total number of repeated code slices (made up of 10 lines), the largest number of repetitions of a single slice, the average number of repetitions per each repeated code slice, and the median number of repetitions. These results were taken from the Platform R19 source code.

Element	Repeated Slices	Maximum Slice Repetition	Mean	Median
DCDM	1524	53	2.89	2
DOPRA	18556	51	2.45	2
RTOS	767	12	2.08	2
VPP	7968	413	3.06	2
VRP	517294	6404	2.85	2

The VRP element had significantly more cases of code repetition than the other elements, although this could be due to the large code base in comparison to the size of the others. Interestingly, the average number of repetitions per code slice was greater in element VPP, suggesting that while there were more code repetitions in other elements, the same code was repeated in more files in VPP.

It should be noted that these are only relative comparisons as a definitive observation could not be made due to the nature of the script used to enumerate repeated slices of code.

Improvement Analysis

Product

The following table compares the total number of repeated code slices for R17 and R19 in order to determine if there was any improvement in code repetition.

As can be seen below, code repetition was reduced by a large percentage in R19, with 82.79% less code repetition in the code base.

Project	Repeated Slices	Comparison Improvement
R17	7,592,788	-
R19	1,306,906	82.79% improvement

This improvement could also be noted in the rest of metrics. Results from R17 can be reviewed below for comparison. The mean value, or in other words, the average number of repetitions of a certain code slice, was reduced from 4.17 instances to 2.63. The median value was also reduced from 3 instances to 2. Lastly, the maximum slice repetition value was reduced from 125,460 instances to 4402; a substantial reduction.

R19 could still benefit from investigating the cause of such a high maximum repetition value. There is a possibility of this value being due to some sort of false positive, in which case increasing the slice size to a larger number of lines, such as 15, could help provide further insight into this issue. If it is not related to false positives it could indicate an extensive repetition of files (see Section 2.1.3 – File Repetition Rate).

Supplemental Data

R17 Data table:

Element	Repeated Slices	Maximum Slices Repetition	Mean	Median
Product R17	7,592,788	125,460	4.17	3

Platform

The following table compares the total number of repeated slices of code for R17 and R19 in order to determine if there was any improvement in code repetition. As shown below, R19 reduced code repetition by 79.52% from R17.

Project	Repeated Slices	Comparison Improvement
R17	2666906	-
R19	546109	79.52% reduction

Comparing R19 with the data gathered from R17, shown below for reference, significant improvements were made in the RTOS element. The mean in RTOS R19 was 2.08, reduced from the R17 mean of 5.23 instances per slice of code. The median for RTOS R19 was 2 instances of repetition for the same slice of code compared to 5 for RTOS R17. This was reduced to 2 instances in R19. In general, there was a broad improvement across all elements as the number of repeated slices of code were markedly reduced from R17 to R19.

Supplemental Data

R17 Data table:

Element	Repeated Slices	Maximum Slices Repetition	Mean	Median
GMDBC401B100	2,873	49	2.76	2
GMDBC430B200	2,392	47	2.65	2
DOPRA	912,961	156	2.96	3
RTOS	6,955	46	5.23	5
VPP	34,222	223	3.92	2
VRP	1,707,503	4,520	2.54	2

2.1.5 Redundant Code

Data Collection

Data for this element was collected in both R19 and R17 for Platform and Product.

Objective

The objective of this element was to identify redundant slices of code and to analyse the change between R17 and R19 for Product and Platform.

Results Assessment

For both Platform and Product of R19, redundant slices of code were almost entirely eliminated, with only one such code slice being identified in the Product code base, and none in the Platform code base.

Change Analysis

For R17, a total of 750 redundant slices of code were identified in Product and a further 155 were found across the different elements of R17 Platform code.

Release	Redundant Slices	Release	Redundant Slices	Change
Product R17	750	Product R19	1	-100%

Platform R17	Redundant Slices	PlatformR19	Redundant Slices	Change
GMDB	0	DCDM	0	-
DOPRA	2	DOPRA	0	-100%
RTOS	8	RTOS	0	-100%
VPP	9	VPP	0	-100%
VRP	136	VRP	0	-100%

2.1.6 Code Maintainability Analysis

The Cyclomatic Complexity metric proposed by Thomas McCabe in 1976[2] produces a quantitative measure of software complexity based on the number of linearly independent paths through some source code. This is calculated based on the number of 'nodes' (N), or processing tasks and 'edges' (E), or control flows, using the formula: V(G) = E - N + 2 (although some variants of the formula have been proposed).

The diagram below illustrates how this is calculated:

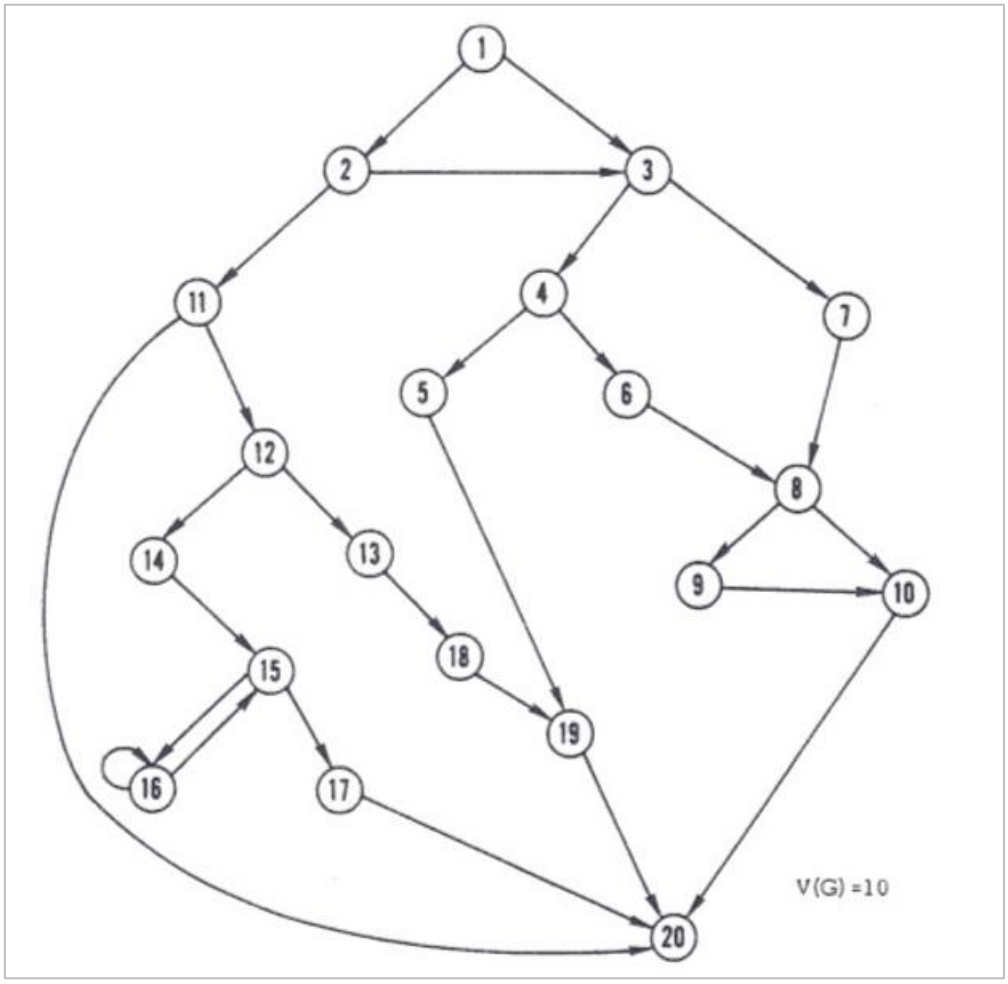

Figure 1: An example of the nodes and edges used to calculate Cyclomatic Complexity

[2]http://literateprogramming.com/mccabe.pdf

The number produced by the Cyclomatic Complexity measure can be used in conjunction with the following table to assess the complexity, and thus the maintainability, of code:

Complexity Number	Meaning
1 - 10	Structured and well written code High Testability Cost and Effort is less
10 - 20	Complex Code Medium Testability Cost and Effort is medium
20 - 40	Very complex Code Low Testability Cost and Effort is high
> 40	Not at all testable Cost and Effort very high

An advantage of the Cyclomatic Complexity measure, when compared with the Lines of Code approach, is that it is less affected by the programming language used as it relies more on the logic of the code. However, a report by David Maxwell, of Coverity, in 2008[3] demonstrated a strong positive correlation between the number of lines of code in a code base and the code's Cyclomatic Complexity suggesting that the degree of difference between the two approaches is perhaps not very significant.

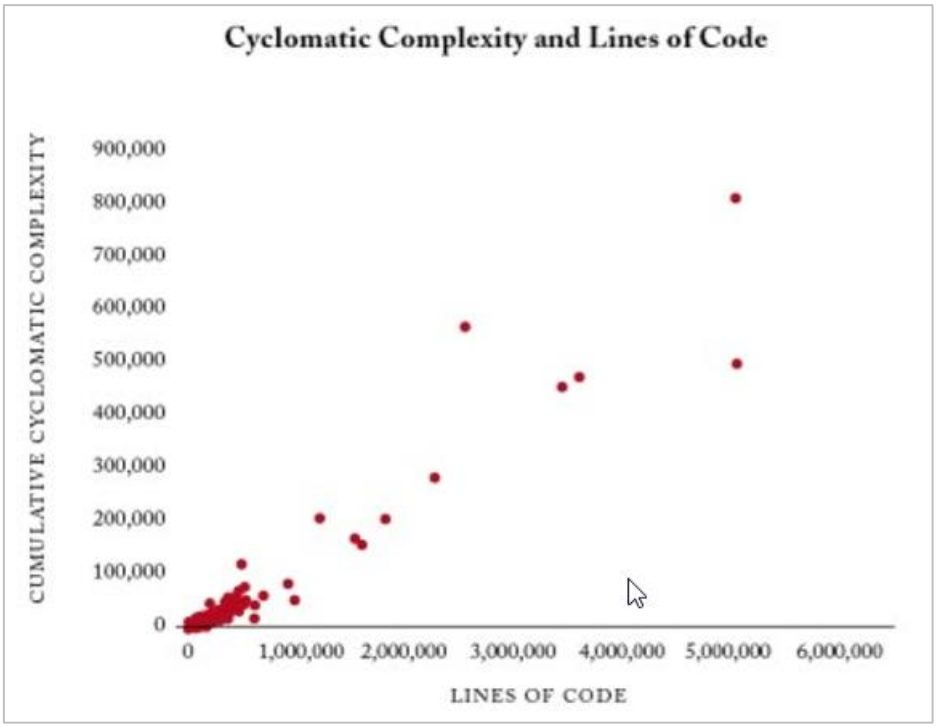

Figure 2: Strong positive correlation between LOC and Cyclomatic Complexity from Technology Innovation Management

[3] https://timreview.ca/article/156

Halstead Complexity Measures

Maurice Halstead proposed the Halstead Complexity measure in 1977. Unlike the Lines of Code and Cyclomatic Complexity measures, the Halstead Complexity approach produces a number of quantitative measures, or metrics, relating to different aspects of code complexity. These metrics are produced according to the number of operands and operators in use within a code base where:

Parameter	Meaning
n1	Number of unique operators
n2	Number of unique operands
N1	Number of total occurrence of operators
N2	Number of total occurrence of operands

Then, the metrics are derived as follows:

Metric	Meaning	Mathematical Representation
n	Vocabulary	n1 + n2
N	Size	N1 + N2
V	Volume	Length * Log2 Vocabulary
D	Difficulty	(n1/2) * (N1/n2)
E	Efforts	Difficulty * Volume
B	Errors	Volume / 3000
T	Testing Time	Efforts / S (where S = 18 seconds)

This approach produces a more granular set of measures than is provided by either the Lines of Code or Cyclomatic Complexity approaches. A detailed discussion of these is beyond the scope of this document. However, they are discussed in some depth here: http://www.virtualmachinery.com/sidebar2.htm.

There is some question about the applicability of the Halstead approach to measuring complexity when examining object orientated (OO) code. Applied at the method level it is considered useful but looking at larger slices of OO code can lead to misleading results. It should be noted that this same reservation can also be applied to the Cyclomatic Complexity measure.

Maintainability Index

The Maintainability Index (MI), comprising some combination of the Lines of Code, Cyclomatic Complexity and Halstead Metrics measures of complexity has become a de facto standard for measuring complexity. As there are different approaches to measuring each of the component metrics there can be slightly differing implementations of the MI. Similarly, there have been slightly different formulas applied to the three constituent metrics in order to arrive at the final MI figure.

The original implementation of the MI resulted in a figure between 171 at the top end of the range and an unbounded negative number using the formula:

```
171 - 5.2 * ln(Halstead Volume) - 0.23 * (Cyclomatic Complexity) - 16.2 * ln(Lines of Code)
```

This was considered somewhat unwieldy and unintuitive by Microsoft which consequently suggested a formula which produced a result between 0 and 100[4]. The (Excel) formula Microsoft proposed is:

```
MAX(0,(171 - 5.2 * ln(Halstead Volume) - 0.23 * (Cyclomatic Complexity) - 16.2 *
ln(Lines of Code))*100 / 171)
```

The following rules were applied to the output (where 'red' is considered overly complex code and 'green' of acceptable complexity):

Maintainability Index	Colour Code
0 - 9	Red
10 - 19	Amber
20 - 100	Green

The use of the MI to measure code complexity can be problematic when comparing differing implementations. As discussed above, Microsoft proposed a standardised approach that is widely accepted and is integrated into their Visual Studio products. Despite these differing implementations there is little evidence to suggest that any one is significantly superior to others relying as they do on more or less identical fundamental concepts and measures.

Of primary importance is the consistent use of a single approach to the calculation of MI across both longitudinal and horizontal comparisons of complexity. In other words, as long as the same approach is used to calculate the MI of a code base between earlier and later versions or between two different code bases the results are likely to be useful. Conversely, comparisons made between MI figures calculated using different approaches are likely to be considerably less useful.

Conclusions

The four approaches to measuring code maintainability/complexity discussed above are all subject to criticism. Nonetheless, they have largely been demonstrated to produce reasonable, reliable and consistent measures. The MI has become a de facto standard in part because of its adoption by Microsoft and consequent inclusion in the very widely used Visual Studio suite of tools. This has similarly led to adoption of this approach into other Integrated Development Environments (IDE).

It is important to bear in mind the relative strengths and weaknesses of the differing approaches and to ensure that a granular view of the code base under inspection is maintained in order to avoid the situation in which, while the overall code base receives a good maintainability score, some modules or functions within it are of a significantly lower quality. To this end implementations of the MI in most major IDEs allow results to be organised both by the differing results of the Lines of Code, Cyclomatic Complexity and Halstead Metrics tests and by module or even individual functions or methods.

While measures of maintainability provide a good means of tracking the performance of development teams and the quality of the code they produce it is also necessary to adopt a proactive approach to good development practices. To this end it is essential that good quality code style guides and standards are produced and that adherence to them is monitored and enforced.

[4] https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/

2.2 Further Specific Code Analysis

2.2.1 Safe and Unsafe Functions

Data Collection

Product and Platform, R17 and R19 data was collected for this work item. These tasks have been bundled together as they were closely related.

Objective

To analyse the overall use of unsafe functions within both the Platform and Product code.

To make a comparison between the Platform and Product code from R17 and R19 and analyse whether or not best practice was followed, where applicable.

Current Assessment

The safe C functions' density was always higher that the unsafe C functions' density in the Platform code base of R19. In addition, the DCDM and VPP modules did not have any calls to unsafe C functions. The total ratio of safe/unsafe C functions' densities in R19 was 6118.65%.

Platform					
Release.Element	**Safe Functions**	**Safe Functions Density**	**Unsafe Functions**	**Unsafe Functions Density**	**Total Lines of Code**
DCDM	966	5.88	0	N/A	164245
DOPRA	4734	8.3	422	0.74	570282
RTOS	118	4.39	113	4.28	26870
VPP	1272	6.15	0	N/A	206925
VRP	116448	10.68	1494	0.14	10904706

Product				
Safe Functions	**Safe Functions Density**	**Unsafe Functions**	**Unsafe Functions Density**	**Total Lines of Code**
170591	10.80	402	0.02	15791341

Improvement Analysis

The following two tables contain the number of safe and unsafe functions found in Platform and Product in both releases – R17 and R19 – along with their relevant densities.

Platform					
Release.Element	**Safe Functions**	**Safe Functions Density**	**Unsafe Functions**	**Unsafe Functions Density**	**Total Lines of Code**
R17.GMDB	883	5.87	0	N/A	150362
R19.DCDM	966	5.88	0	N/A	164245
R17.DOPRA	18606	9.01	422	0.2	2065296
R19.DOPRA	4734	8.3	422	0.74	570282
R17.RTOS	338	7.95	221	5.20	42506
R19.RTOS	118	4.39	113	4.28	26870
R17.VPP	3097	10.19	481	1.58	303987
R19.VPP	1272	6.15	0	N/A	206925
R17.VRP	89575	5.98	7649	0.51	14978489
R19.VRP	116448	10.68	1494	0.14	10904706

The density of safe functions within the Platform code base was always higher than the unsafe functions discovered, proving a noticeable improvement from R17 to R19 in adhering to security best practice.

Although the DOPRA, RTOS and VPP modules had lower safe functions density in R19 compared to R17, they were still higher than the unsafe functions density.

In addition, code was substantially optimised in R19, reducing the total number of calls to C functions. For instance, the DOPRA module was optimised reducing the total number of C functions by 73.90% while the number of unsafe C functions did not increase.

Note that, although the VPP module's safe functions density decreased in R19, this was due to an important optimisation of the code base, reducing the total number of calls to C functions in 64.45%. In addition, no calls to unsafe C functions were found in the VPP module in R19.

The ratio of safe/unsafe C function usage in the R19 Product code base was a considerable improvement on the same ratio in the R17 code base. See the table below:

Product					
Release	**Safe Functions**	**Safe Functions Density**	**Unsafe Functions**	**Unsafe Functions Density**	**Total Lines of Code**
R17	6048	0.17	24563	0.69	35795304
R19	170591	10.80	402	0.02	15791341

In the case of the Product code base, a great improvement was noted in the use of safe C functions in R19, bringing the safe C functions density from 0.17 to 10.80 which in relative terms translated to 6252.94% increase of the density of safe C functions. At the same time, the use of unsafe C functions decreased from 24563 to 402, which is a reduction of 97.10% of the density of unsafe C functions from R17 to R19.

References

- Bounds-checking Interfaces: Field Experience and Future Directions (2018, Robert Seacord) http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2336.pdf
- ISO/IEC Programming Languages – C 4th edition (ISO/IEC 9899:2018)
- Secure Coding in C and C++ - 2nd edition (2013, Robert Seacord)

2.2.2 Usage of Safe Security Functions

Data Collection

Product and Platform, R17 and R19 CodeMars static analysis results were collected for this work item.

Objective

To review the results of running the Huawei CodeMars static analysis tool across R17 and R19.

To analyse the quality of the diagnostic output of the tool.

To discuss how best to utilise the results of static analysis tool output.

Current Assessment

CodeMars is a tool developed internally by Huawei for static analysis of source code. CodeMars appeared to be able to handle various programming languages and performed various kinds of static analysis. This analysis focused on the output of CodeMars specifically when instructed to check for issues relating to insecure C coding practices.

The tool specifically targeted use and abuse of safe and unsafe functions. It reported issues, which seemed specific to the sorts of issues Huawei discovered while attempting to move to using safe functions. The tool also specifically targeted use of Huawei's implementation of secure C functions.

The errors reported by the tool had few obvious false positives. The majority of the errors which were analysed were not false positives but would require further manual analysis to verify. This indicated that Huawei's internal CodeMars tool produced useful and meaningful error reports for their code base and could be used as an indicator of Huawei's progress in the implementation of secure C coding practices from R17 to R19.

A large fraction of the warnings reported by the tool did not indicate bugs when further investigated. This is normal for most static analysis tools. However, these instances were mostly examples of situations where code could be further clarified to improve maintainability and avoid the accidental introduction of bugs which would affect the security of Huawei's code. As such, the warnings, despite not always exposing bugs, are a useful and meaningful tool for analysing and improving the code base.

The results of the R19 analysis indicate a relatively low number of warnings and errors in both Product and Platform code bases. Specifically, the tool reported 9 errors and 1803 warnings in the entire Platform code base and 46 errors and 1807 warnings in the entire Product code base. This gives a density of less than 1 error per 1 000 000 lines, and about 3 warnings in every 20 000 lines for Platform. Furthermore, for Product the results indicate a density of just under 3 errors per 1 000 000 lines, and about 2 warnings in every 20 000 lines.

Areas where the tool could be improved include the ability to selectively silence certain warnings for the few situations a false positive is found and where refactoring the code to prevent CodeMars from highlighting the issue would be a detriment to the code base. There would need to be a strict process implemented to review uses of these exclusions when they are added to the code base to ensure they are not abused. The main reason for this measure is that in some situations false positives can force programmers to write their code in a way, which tries to remove the warning rather than trying to write the code in a way, which is readable, simple and safe. It is especially important to ensure such facilities are available before any policies which prevent programmers from contributing code which introduces warnings are implemented, as it is in just these situations that programmers may write substandard code in order to avoid a false positive.

Additionally, the tool could easily be expanded to try to encourage programmers to standardise on how sizes of objects are passed to the secure C functions. This would further clean up the code and avoid classes of bugs related to refactoring pieces of code, which deal with types and memory management.

Improvement Analysis

The results indicated that R19 improved over R17 when it comes to the number of errors and warnings reported by CodeMars. This is most likely in part due to Huawei's focus on simplifying and restructuring the R17 code base to remove redundant and duplicate code. However, Huawei also focused on implementing more secure C coding practices in R19.

In general, it was noted that the issues which CodeMars targeted, were issues which the R17 code base presented. Hence, CodeMars found and reported issues which, when fixed, would positively impact the R17 code base. It was found that a good number of the issues the tool reported in R17 were fixed in R19 and the analysis of a sample of the issues found no evidence of widespread attempts to obfuscate or hide problems and instead found that the problems had been fixed in an effective manner.

Platform

Error	R17 Count	R19 Count	Change (%)
outOfBoundsRead	983	0	-100%
outOfBoundsWrite	16	4	-75%
paraNotUsed	9	3	-67%
paraNotUsedInMacro	13	0	-100%
sameSecondAndFourthParas	16	2	-88%
sameSecondAndFourthParasInMacro	2	0	-100%
Total:	**1039**	**9**	**-99%**

Warning	R17 Count	R19 Count	Change (%)
destMaxIsNumber	2236	13	-99%
falseDefinedSafeFunc	5	0	-100%
funcRetNotCheck	18344	1187	-94%
retValueNotCheck	2480	54	-98%
safeFuncDestMaxNotAsDefined	951	495	-48%
safeFuncRedefined	98	15	-85%
safeFuncWithParaRedefined	935	7	-99%
selfDefinedRiskyFunc	279	29	-90%
selfDefinedSafeFunc	510	3	-99%
Total:	**25838**	**1803**	**-93%**

Product

Error	R17 Count	R19 Count	Change (%)
outOfBoundsRead	744	34	-95%
outOfBoundsWrite	28	11	-61%

Error	R17 Count	R19 Count	Change (%)
paraNotUsed	1	1	0%
paraNotUsedInMacro	12	0	-100%
sameSecondAndFourthParas	137	0	-100%
sameSecondAndFourthParasInMacro	37	0	-100%
Total:	**959**	**46**	**-95%**

Warning	R17 Count	R19 Count	Change (%)
destMaxIsNumber	4440	4	-100%
falseDefinedSafeFunc	4	0	-100%
funcRetNotCheck	4730	1753	-63%
retValueNotCheck	48	0	-100%
safeFuncDestMaxNotAsDefined	1957	9	-100%
safeFuncRedefined	586	13	-98%
safeFuncWithParaRedefined	1987	2	-100%
selfDefinedRiskyFunc	809	26	-97%
selfDefinedSafeFunc	157	0	-100%
Total:	**14718**	**1807**	**-88%**

2.2.3 Open Source and Third Party Software

Data collection

Data for this analysis was collected from R19 in Platform and Product using a mixed automatic and manual process. A combination of FOSSID and manual inspection were used to eliminate false positives

Objective

To investigate the use of open source and third party dependencies within the source, and identify any areas of multiple copies, or multiple versions which need eliminating.

Current Status Assessment

For Product in R19, no duplicate versions of open source dependencies were identified. Additionally, it should be noted that between Product R17 and R19, all open source dependencies have been consolidated within the *open_source* directory, improving auditability and development accessibility for other modules within the firmware to be built against a single canonical version for each package, as well as tracking dependencies. This was not the case in the R17 build, for which there was not sufficient time during assessment to recover a listing of all open source packages.

There were many more packages in Platform R19, some of which appeared to be duplicated.

Package	Version	Version at Release Time
linux	4.4.51 patched to 4.4.171	4.4.178
yaffs2	1f3d64d1d804fef 6715126dead54bf 30b176c67e	
lzma (7zip)	18.05	
acl	2.251	
attr	2.4.46	
audit	2.8.1	
base-passwd	3.5.45	
bash	4.2	
binutils	2.31.1	
busybox	1.29.2	
bzip2	1.0.6	
cifs-utils	6.2	
coreutils	8.22	
cracklib	2.9.0	2.9.7
cracklib	2.8.19-1.debian	
crash	7.2.0	
cronie	1.4.11	
curl	7.61.0	
dhcp	4.4.1	
dosfstools	4.1	
e2fsprogs	1.42.9	
elfutils	0.17	

Package	Version	Version at Release Time
ethtool	4.8	
expat	2.1.0	
flex	2.5.37	
gettext	0.19.8.1	
glib	2.54.2	
glibc	2.27	
gmp	6.1.2	
grep	2.2	
gzip	1.1	
iproute	4.11.0-0.el7	
iptables	1.4.21	
json-c	0.11-20130402	
eppic	50615	
kdump-anaconda-addon	003-23-g80e78fb (80e78fb4b2529e 6fbde6dce8d5299 1fd08ad36a9)	
kexec-tools	2.0.15	
makedumpfile	1.6.2	
kmod	20	
kpatch	0.4.0	
less	458	
libaio	0.3.109	
libcap-ng	0.7.5	
libcap	2.25	

Package	Version	Version at Release Time
libcgroup	0.41	
libestr	0.1.9	
libevent	2.1.8-stable	
libfastjson	0.99.4	
libffi	3.0.13	
libhugetlbfs	2.21	
libnl	3.2.28	
libpcap	1.5.3	
libselinux	2.5	2.9
libsemanage	2.5	2.9
libsepol	2.5	2.9
libtirpc	0.2.4	
libtool	2.4.2	
Libusb-compat	0.1.4	
libusb	1.0.21	
libxml2	2.9.8	
logrotate	3.8.6	
lsof	4.87-rh	
ltrace	0.7.91	
LVM	2.2.02.177	
boom	0.8.5	
lzo	2.06	
mtd-utils	2.0.2	
musl	1.1.18	
ncurses	5.9	
net-tools	2.0.20131004git	
netbase	5.4	
nfs-utils	1.3.0	
openssh	7.9p1	
openssl	1.1.1	
LinuxPAM	1.3.1	
pciutils	3.5.1	
pcre	8.42	
policycoreutils	2.5	

Package	Version	Version at Release Time
sepolgen	1.2.3	
popt	1.13	
procps-ng	3.3.15	
psmisc	22.2	
quota	4.01	
readline	6.2	
rpcbind	0.2.0	
rsyslog	8.24.0	
sed	4.2.2	
shadow	4.1.5.1	
squashfs	4.3	
strace	4.12	
sysstat	10.1.5	
tcp_wrappers	7.6-ipv6.4	
javazic	1.8-37392f2f5d59	
tzcode	2018f	
tzdata	2018f	
u-boot	2018.07	
usbutils	7	
ustr	1.0.4	
util-linux	2.23.2	
xinetd	2.3.15	
xz	5.2.4	
zip	39	
zlib	1.2.11	
build tools:	build tools:	
autoconf	2.69	
automake	1.15.1	
opkg-utils	0.3.6	
pkg-config	0.29.2	
pseudo	1.9.0	
sqlite	3220000 (3.22.0)	
yocto/poky	sumo-19.0.1	

2.2.4 Security Compilation Options Evaluation

Data collection

Data for this analysis was collected on Product R19 and Platform R17 and R19 binary packages with some exceptions explained in the Caveats section.

Objective

To review the compilation options set in binaries of the R17 and R19 binary packages.

Current Status Assessment

The most noticeable piece of data was the extensive use of the NX bit across all binaries in both Platform and Product. There was also an increase in the use of PIE/PIC option for RTOS and VPP packages in R19. Symbols Tables were correctly stripped out in the majority (82%) of the Product binaries.

RELRO

Relocation Read-Only is a feature, which marks relocation sections as read-only. In the "Full" version of this, (where executables are linked with the `-z now` flag) this is fully done before the start of execution.

Using the Full Relro feature may result in some performance reduction in the case of large executables with many imported libraries.

There was a noticeable increase in the use of Relro in R19. The combination of Partial Relro and Full Relro added up to 100% coverage. The majority of R19 packages had a high ratio of Full Relro usage (96.37% onwards).

Stack Canary

Stack Canaries are used as a mechanism to detect stack-based overflows and increase the difficulty of exploitation by placing a token at the end of stack frames. When overwritten with another value, an exception will be raised causing the execution to stop. Stack Canaries are not supported on some architectures (including ia64, alpha, mips and hppa).

The analysed packages for R19 Platform and Product had a good Stack Canary usage ratio. R19 Product packages had an ~82% Stack Canary usage and Platform was ~88%. There are situations were GCC compiler refuses to apply Stack Canaries for trivial functions due the simplicity of some functions.

NX Bit

The NX Flag indicates that code execution has been banned from data memory pages, such as the stack and the heap. This mitigation makes exploitation of buffer overflows vulnerabilities far more challenging forcing the attacker to use more specialised techniques like Return-Oriented Programming (ROP). There are instances where this flag may cause functionality issues for executables – specifically if the executable use self-modifying code patterns. This flag also depends on operating system support.

All the analysed packages for R19 had a 100% usage ratio of the NX bit. No improvements were needed for R19.

Position Independent Executable (PIE) and Position Independent Code (PIC)

The PIE flag means that an executable is safe for relocation using ASLR (Address Space Layout Randomisation). In order to be effective for security, this flag also needs to combined with ASLR being enabled

in the kernel. ASLR makes exploitation of overflow attacks more challenging by adding a randomised offset to the addresses of functions and data at runtime.

The analysed packages for Product R19 showed good use of the PIE/PIC option, with 98.64% coverage. The same applied for Platform R19, with the exception of RTOS which had 50.16%. coverage. This situation was a limitation of the version of Checksec used for this assessment since it did not correctly detect the use of the PIC option in Shared Object ELF files.

RPATH and RUNPATH

RPATH and RUNPATH introduce an increased risk of runtime dependency loading attacks in Linux-based environments, enabling an attacker to potentially load malicious code replacing expected functionality at runtime.

RPATH and RUNPATH were correctly disabled in more than 95% of binaries for R19 on Product and Platform. This showed that Huawei had made noticeable progress from R17 to R19.

Symbol Tables

Symbol table information for executables facilitates debugging by allowing the identification of function names. This can facilitate reverse engineering by third parties, and it is recommended that executables should be stripped of symbols.

The vast majority of binaries in Product were correctly stripped and did not have any symbol information (~82%). During the Platform build process, no binaries were stripped as these were stripped later in the process of linking and compiling the Product code. The presence of symbols in Platform binaries for debugging purposes was an expected situation in the build process as they were only used to build the Product binaries.

Improvements Analysis

Overall, there was a widespread and visible improvement in the use of the security compilation options from R17 to R19, especially with regard to Relro flags and Stack Canaries. The use of the PIE/PIC option also increased.

Table of Findings

	RELRO		Stack Canary	NX Bit	PIE/PIC	No RPATH	No RUNPATH	Symbols Stripped **	# Files
PRODUCT									
R19 MPLX	**Partial:** **Full:**	10.67% 89.32%	82.50%	100%	98.64%	99.84%	96.47%	82.83%	1246
R19 MPSC	**Partial:** **Full:**	10.73% 89.27%	82.42%	100%	98.63%	99.84%	96.45%	82.74%	1240
This space intentionally left blank									
PLATFORM									
R19 DOPRA	**Full:**	100%	100%	100%	100%	100%	100%	N/A	3
R19 RTOS	**Partial:** **Full:**	0.40% 99.60%	83.14%	100%	50.16%*	100%	99.88%	N/A	2486
R19 VRP	**Partial:** **Full:**	3.76% 96.37%	91.54%	100%	99.09%	100%	98.49%	N/A	331
R17 DOPRA	**Partial:**	100%	100%	100%	100%	100%	100%	N/A	3
R17 RTOS	**No:** **Partial:** **Full:**	29.02% 69.66% 1.32%	1.68%	100%	85.23%	96.13%	99.93%	N/A	1368
R17 VRP	**Partial:**	100%	0%	100%	0%	100%	100%	N/A	6

* The PIC security option value could not be accurately determined using the Checksec tool due to the nature of this header which is not included in ELF file header (this was agreed at the start of testing). Checksec uses these headers to enumerate the binary security options and therefore could not enumerate files with the PIC option set. In order to overcome this limitation a custom script was created to attempt to locate files with the PIC flag. Unfortunately, it was not possible to enumerate all binaries with the PIC option.

** Symbols Tables for Platform were considered irrelevant because they were only used during the compilation process to generate the Product binaries. So this is an acceptable result for this analysis.

R19 DCDM, R19 VPP and R17 GMDB packages were not included in the above table because they only had intermediate objects which had not been linked yet.

2.3 Build Process Engineering

2.3.1 Build Without Root Privilege

Data collection

Data for this analysis was collected on the build machines (two SUSE 12.4 hosts) focusing on ensuring that the non-root ‘huawei’ user was not able to elevate their privileges without supplying appropriate credentials.

Objective

To review the host build configuration to detect possible methods for privilege escalation exploitation and other common problems.

Current Assessment

Overly permissive sudoers file configuration

The Product build environment was configured in such a way that the ‘huawei’ user could gain root privileges without having to provide a password. Consequently, controls in place intended to restrict the actions of the ‘huawei’ user could be trivially bypassed.

It was also confirmed that it was only possible to build the R19 code base by using the root user.

The sudoers file contained a NOPASSWD entry for the ‘huawei’ user for all commands, resulting in that user being able to execute any command with the privileges of the root user without being required to enter a password.

Although this configuration is often used in development environments for the convenience of system administrators, it is bad security practice.

This configuration considerably undermines many of the controls implemented by the operating system to protect the confidentiality and integrity of all aspects of the system and is not appropriate to a production environment.

Discussions with Huawei indicated that an Ansible playbook would be provided for the next release environment and this would eliminate the need for a sudoers file. However, this had not been implemented for R19.

2.3.2 Reproducible Builds

Data Collections

Data was collected on the Platform and Product R19 and R17 source code, compilation guides, and final compilation outputs.

Objective

The objective of this element was to evaluate the build process and ability to create reproducible builds for R19 on both Platform and Product. Once the R19 process was studied, it was then compared to R17 to make note of any improvements or persistent issues.

An important point of focus for this element was analysing the ease and simplicity of successfully creating and comparing reproducible builds by following the provided documentation.

Current Assessment

Environment

Product R19

The build environment for R19 only required one SUSE Linux virtual machine image which appeared to be set up and ready to be used for compilation without any additional configuration on the part of the client. Minimising the amount of set up needed in order to carry out compilation allowed Huawei to eliminate a number of potential factors that could to lead to variations or errors in the expected output.

If this was only the case in the testing environment, it is recommended that an equivalent image or snapshot be provided to the client with the environment and tools already preconfigured.

Platform R19

Two SUSE Linux virtual machines were required for compilation of Platform R19. Like the Product build environment, no additional set up was required in the testing environment. However, unlike Product, Platform R19 did include instructions for configuring the virtual machines. It is therefore possible that the environments had been preconfigured to facilitate testing (removing the need for configuration by the consultants). If it is intended that the environments should be delivered preconfigured, it is recommended that the documentation be updated to reflect this.

In general, it is recommended that an image or snapshot is provided to the client with the environment and tools already preconfigured in order to reduce the likelihood of variables being introduced that could cause unexpected compilation output results. If this is not possible, then the environment set up process should be reviewed with a view to simplifying it as far as is practical; the documentation should then be updated in line with the configuration.

Compilation

Product R19

The compilation process for R19 was well documented and easy to follow, only requiring the client to copy the source code to the correct folder, set the proper permissions, and launch the build script.

Small improvements could be made to the documentation to increase readability. For example, consistency in how the instructions are presented as well as emphasising the importance of following the guide exactly, particularly when reproducible builds are a requirement.

Platform R19

The compilation instructions for Platform R19 consisted of very few steps, mostly related to setting up the folders, uncompressing the source, and running the build script. Like the Product environment, following the instructions was critical to obtaining a reproducible build.

Build Results

Product R19

In Product R19, it was possible to create a reproducible build by following the process outlined in the compilation guide. Although, due to the nature of the secure testing environment, a small number of additional steps were needed that were not documented. Discussions with Huawei confirmed that these additional steps would not be required by the client.

While R19 did compile reproducible builds, the documentation for making the comparison with other builds could be improved. The lack of clarity in this documentation, could lead to client compilation results appearing to be different from the comparison results provided by Huawei. The unpacking process instructions of the comparison package and recently compiled package did not show the same clarity as the documentation for the compilation process.

Additionally, no suggestions were made on how to compare the results or which results should match and which should not. This would not be a problem if the client only had to compare the packages themselves, or if all of the unpackaged files matched. In this case, R19 had two files that did not match, a signature file, and a file containing a list of all files. These details should be documented in the compilation guide and if possible automated in a script.

Platform R19

R19 was confirmed to be reproducible after uncompressing the archived results files and comparing their contents against another compilation. It was not possible to compare the compressed files themselves as the calculated hashes did not match. While this is a common occurrence for a number of reasons such as archive time stamps, sort order complications, user/group IDs, solutions do exist to stabilise the creation of compressed archives. For example, if it is the compressed archive's time stamp that causes the change in the expected hash value, this could be remediated using the "--mtime" option for ".tar" files. There are additional methods for other compressed archive types.

Improvement Analysis

Significant improvements were found in the build process from R17 to R19 in both Product and Platform. In general, all aspects of the build process were simplified, including the environment, which required a significantly reduced number of virtual machines for compilation and the documentation, which was simplified and easier to follow. Improvements such as these allow the client to build the source code in less time and with more confidence that the resulting package will be correct.

In the time allotted for testing it was not possible to successfully compile R17 for Platform or Product. It was not clear if this was because of the complexity of the build environment, outdated instructions, or the testing environment.

Final Thoughts on Reproducibility of R19

While it was possible to create reproducible builds, it was required that users use a preconfigured virtual machine and follow all compilation instructions, including compiling each of the components in their respective directories. Additionally, it was required to use a specific time stamp in Product (a time stamp taken from a specific server's local time had to be set into a certain configuration file before the build script could be run). Any deviation from this could result in a non-reproducible build due to a number of factors such as directory paths, environment variables, time stamps, etc.

It is acknowledged that the time stamp requirement was a temporary fix to a specific test environment issue. It is understood from discussions with Huawei that this was addressed after the assessment was completed.

If Huawei would like to allow users more flexibility, further work and testing is recommended in order to ensure that reproducible builds can be compiled independent of environment variables. This would require extensive testing to ensure that both input and output when compiling remain stable.

Below is a list of common factors that affect reproducible builds:

Time Stamp	Time Zone	Locale
Shell	Hostname	Environment variables
UID	GID	Umask
File owner	Build path	File system ordering
CPU model	Kernel version	C compiler version
Linker version		

Each of these should be addressed in the future if flexibility is desired for the build process. One reason to strive for this kind of flexibility is that it allows clients to work in environments they are more comfortable with or conform with their own internal policies. Another benefit of allowing clients to produce reproducible builds in their own environments would be the transfer of responsibility for the security of the environment from Huawei to the client. Currently Huawei is responsible for this aspect. Having a flexible reproducible build also removes the likelihood that slight deviations from the compilation process or environment setup will result in a non-reproducible build.

These recommendations for future work have only been reported to provide ideas for further streamlining the reproducible build process. The build process for R19 was easy to follow; use of the correct environment and following the instructions resulted in reproducible builds.

2.3.3 Compiler Warnings

Data Collection

Product and Platform, R17 and R19 build logs were collected for this work item.

Objective

To analyse the reduction in the number of build warnings generated between R17 and R19.

To analyse some of the ways in which this reduction may have been achieved.

To advise on best practice and suggest further improvements.

Current Assessment

The analysis of R19 concluded that warnings (at least -Wall or better) were enabled for most if not all parts of the build process. The only exception may be a relatively insignificant part of the code base comprising a few subprojects in RTOS, although this could not be definitively confirmed within this assessment.

The Platform code base had no warnings reported during the build process. It did not appear that any systematic attempts were made to silence warnings with the notable exception of the use of `(void)` casts to attempt to silence errors regarding ignored function results. Another issue noticed in the code base was an over-reliance on pointer casts which may hide hard to track down issues surrounding pointer aliasing and pointer types. In general, it should be noted that through the use of the `void *` type C programmers should not need to rely on casting pointers in most normal and correct code.

It was noted that in some select and localised instances in both the Platform and Product code bases some warnings were being disabled using `-Wno-` flags.

During inspection of the source code for the secure C library, it was noted that the library made an unsuccessful attempt to use gcc's `__attribute__((format))`. The attribute was mentioned inside the source code, but flags were not found which would enable these attributes. However, it should be noted that, this was only the result of a static analysis and could not be completely verified with the source. It would also be pertinent to mention gcc's `__attribute__((warn_unused_result))` which may be useful in the secure C library to allow more warnings to be generated for the misuse of the secure C library. However, this may in some cases introduce false positives, which tools such as CodeMars successfully avoid. The R19 Platform DCDM, DOPRA, RTOS and VRP modules could benefit from having more warnings enabled as these would allow further improvement of those code bases. The Product code base and the Platform RTOS module could benefit from making it more clear which warnings are enabled and for which parts of the build.

R19 Platform Results

Module	Flags	Warnings
DCDM	-Wall	None
DOPRA	-Wall	None
RTOS	-Wall (Partial)	None
VPP	-Wall -Wcast-align -Werror -Wformat=2 -Wpointer-arith -Wtrampolines	None
VRP	-Wall -Wno-builtin-macro-redefined	None

R19 Product Results

Module	Flags	Warnings

mplx_51	-Wall	4
mpsc_52	-Wall	3

Improvement Analysis

Overall, the number of warnings emitted during the compilation process decreased in R19. The Platform code base build completed without warnings in R19, while only 5 warnings were found during the build process for R17. The Product code base also had less warnings as they decreased from 82 in R17 to just 4 (unique) warnings in the R19 build process.

The warning flags enabled in R19 are also a general improvement over R17.

R17 Platform Results

Module	Flags	Warnings
GMDB...SPC401B100	None	None
GMDB...SPC430B200	-Wall	None
DOPRA	-Wall	None
RTOS	-Wall (Partial)	None
VPP	N/A[5]	N/A
VRP	-Wall	5

R17 Product Results

Module	Flags	Warnings
N/A	-Wall	82

[5]No log information was collected for R17 VPP as a full build was never achieved. No analysis could be performed.