# Effective Source Code Analysis with Minimization

Geet Tapan Telang
Hitachi India Pvt. Ltd.
Research Engineer
Bangalore, India
Email: geet@hitachi.co.in

Kotaro Hashimoto
Hitachi India Pvt. Ltd.
Software Engineer
Bangalore, India
Email: kotaro.hashimoto.jv@hitachi.com

Krishnaji Desai
Hitachi India Pvt. Ltd.
Researcher
Bangalore, India
Email: krishnaji@hitachi.co.in

***Abstract*—During embedded software development using open source, there exists substantial amount of code that is ineffective which reduces the debugging efficiency, readability for human inspection and increase in search space for analysis. In domains like real-time embedded system and mission critical systems, this may result in inefficiency and inconsistencies affecting lower quality of service, enhanced readability, increased verification and validation efforts. To mitigate these shortcomings, we propose the method of minimization with an easy to use tool support that leverages preprocessor directives with GCC for cutting out ifdef blocks. Case studies of Linux kernel tree, Busybox and industry-strength OSS projects are evaluated indicating average reduction in lines of code roughly around 5%-22% in base kernel using minimization technique. This results in increased efficiency for analysis, testing and human inspections that may help in assuring dependability of systems.**

## I. Introduction

The outgrowth of Linux operating system and other real-time embedded systems in dependable computing has raised concern in possible areas such as mission critical system. The size of code has grown to approximately 20 million lines of code (Linux) and with this scale and complexity, it becomes impossible to follow traditional methods for meeting the safety and real-time requirements.

Since tools for analysis and testing are getting advanced, the safety and time requirements can be verified and validated by capturing evidence and justifications. Most of these tools are interested in meeting the coverage expectations in terms of code, execution times, resources, throughput and fault tolerance that define the overall dependability of systems.

Code coverage with different analysis and test tools becomes the major part of verification and validation, in order to perform this effectively, we propose method of minimization devised for keeping functional safety and real-timeliness into consideration for narrowed search space verification, false positive reduction, easier human inspection and shorter verification time. The term minimization as shown in figure 1 signifies removal of unused piece of code comprising of `#ifdef` and `#if` blocks. The target code along with configuration file when executed using minimization process produces compilable code without `#ifdef` and `#if` block and other unused lines of code, which is different from execution with GCC preprocessor where compiled code comprises of `#ifdefs`.

The evaluation is exercised on targets such as Linux Kernel source tree, BusyBox [1] tree and similar quantification of other OSS projects.

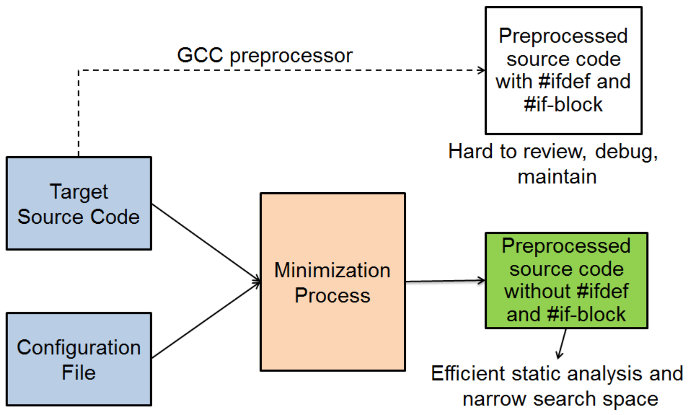

GCC preprocessor
Preprocessed source code with #ifdef and #if-block
Hard to review, debug, maintain
Target Source Code
Configuration File
Minimization Process
Preprocessed source code without #ifdef and #if-block
Efficient static analysis and narrow search space

Fig. 1: Overview of minimization.

## II. Background

In OSS software domain, there are many developers contributing towards the common goal such as real-time, safety-mitigation etc. because of which, the source code developed lacks strict guidelines. Although, there are checks made with semantic patches [2] and other utilities before the source code is committed, no clear coding guidelines are followed that will make the source code easy to inspect and analyze.

Primary problem with the OSS code in embedded domain is the usage of pre-processor directives and conditional code compilations that are used due to varying configuration options. In case of Linux kernel alone, there are more than 10,000 different configuration flags that have to enabled/disabled and thereby used as part of the pre-processor directive in the source code [3].

The configurations of OSS code is easy to enumerate and apply depending on the configuration flags and configure command. However, the source code is still having all of conditional compilation code with pre-processor directives. Too much of conditional compilation code based on configuration, is difficult to inspect and analyze for different static analysis tools. As the configuration options increase, the usage of same in code also increases significantly resulting in analysis complexity.

To solve this problem, there are few tools that are built with pre-processor directives awareness so that during the source code analysis these tools read system configuration and directives to selectively analyze relevant code and skip the disabled code automatically. Some of these tools are GNU cflow, Coccinelle etc. Problem here is that dead code elimination is not the main purpose of these tools. Pre-processor handling is best done with its corresponding compiler in place. Standalone tools cannot do a good job with this as they do not have required constructs configured for effective application of conditional compilation. Hence, a standardized method needs to be made available that can suffice minimization agnostic to other static analysis tools and human inspection. However, the proposed method needs to leverage compiler technology that can best apply the pre-processor directives.

For this purpose, the GCC [4] compiler based pre-processor is selected for realizing the minimization technique. GCC is the choice due to its immense usage in OSS community including Linux, Busybox etc.

### A. *Linux kernel configuration*

Configuration plays a very important role in building any software. In case of OSS, it becomes an essential pre-requisite as the software is developed by different developers with multiple configurations concurrently. To illustrate the same, Linux kernel is a classic example for showing the varied configuration it supports.
The Linux kernel configurations are available in `arch/*/configs`. To alter configuration, integrated options such as `make config`, `make menuconfig` and `make xconfig` are available. Once configuration is done, it gets saved in `.config` file. The indication available in `.config` file has option `=y` illustrating driver is built into the kernel, `=m` for built as a module or it is not selected [5]. The `.config` file appears as below:

```
#General setup
CONFIG_INIT_ENV_ARG_LIMIT=32
CONFIG_CROSS_COMPILE=""
# CONFIG_COMPILE_TEST is not set
CONFIG_LOCALVERSION=""
# CONFIG_LOCALVERSION_AUTO is not set
CONFIG_HAVE_KERNEL_GZIP=y
```

### B. *GCC preprocessor*

GCC [4] is the compiler choice that is used from utils to operating system level and rigorously tested for several years using tools such as CSMITH [6]. Hence, configuration based pre-processor is best applied with GCC and is choice for realizing our methodology.
The GCC preprocessor implements macro language that is utilized to change C programs before they are compiled. The output is similar to input however, the preprocessor directive lines are replaced with blank lines and spaces are appended instead of comments based on the configuration. Certain time some directives may be duplicated in output of the preprocessor, majority of these are `#define` and `#undef` that contains certain debugging options [7].

## III. Related work

The proposed minimization methodology improves static analysis efficiency and easier code inspection. As per prior work regarding source code stripping [8], the GCC options are used to tweak pre-processor directives such that conditional compilation code is stripped as per enabled configurations. This helps in generating .c code that has conditional directives applied to remove the redundant code. Limitation here is that, the approach is not generalized for complete source code tree.

One alternative can be Cflow [9], [10] a GNU based tool, which can preprocess input files before analyzing them and it is integrated with pre-processing option itself, however it renders difficulty because all the required preprocess options needs to be copied and pasted for execution of Cflow leading to incorrectness.

Other alternative can have GCC compilation log and then tweak the log options for each file with required pre-processor options using GREP. Based on which the required .c and .h files with the stripped code based on the pre-processor options can be generated. In subsequent sections, an approach is proposed with Makefile integration and subsequent post processing for easier code inspection and narrowed down search space.

## IV. Minimization Approach

### A. *Minimization Process*

Minimization approach emphasizes on a collection of processes which tweaks integrated MakeFile options to produce compilable minimized code.

*1) Definition:* The term minimization signifies an efficient way to get a set of stripped source code, where all the code which is not required according to `.config` file is left out. Often it is observed that it becomes hard to debug, maintain and verify the code because of macros and preprocessor directives that are expanded during compilation through GCC. This difficulty is subdued with our approach where target source code is free from selected preprocessor options and macros expansion, thereby reducing source code. The approach that has been used for minimization of source code follows the use of GREP command. This filters GCC (used compiler commands) and generates the source tree consisting of limited or useful internal components which are available in shortlisted configuration `.config` file as dedicated output.

The GCC compiler: In general scenario the source code modules are compiled with certain GCC options to make them work [11]. Typical GCC option looks like given snippet of kernel modules:

```
gcc -Wp,-MD,
arch/x86/tools/.relocs_32.o.d
-Wall -Wmissing-prototypes
-Wstrict-prototypes -O2
-fomit-frame-pointer -std=gnu89
-I./tools/include -c -o
arch/x86/tools/relocs_32.o
arch/x86/tools/relocs_32.c
```

For reduction of unused code the above illustrated options are further added with `-E -fdirectives-only` to produce human readable output for easier review, debug, maintain and verify.

*2) Problem:* The major difficulties with GREP based approach is illustrated below:

- It requires a complete build in advance to obtain full set of used GCC commands written in build log.
- The text parsing (grep and gcc commands) is required and has to be acquired from the build log.
- Finally source code needs to be modified to remove `#include` lines.

However with the help of minimization approach code reduction can be achieved by executing `minimize.py` script which requires no pre-build, no build log parsing and no code modification.

The minimization approach is implemented using python script with a stripping technique [12] as below:

- Elimination of configuration conditionals such as `#ifdef #if #endif`.
- Preservation of `#define` macros.
- Preservation of `#include` sentences.

The stripping is initiated and exercised by initially focusing on Linux kernel source code through tweaking the GCC preprocessor options for complete kernel source tree.

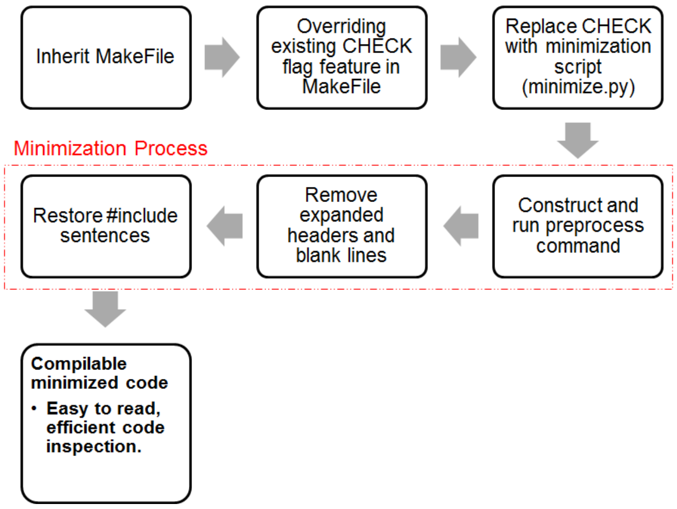

Inherit MakeFile
Overriding existing CHECK flag feature in MakeFile
Replace CHECK with minimization script (minimize.py)
Minimization Process
Construct and run preprocess command
Remove expanded headers and blank lines
Restore #include sentences
Compilable minimized code
• Easy to read, efficient code inspection.

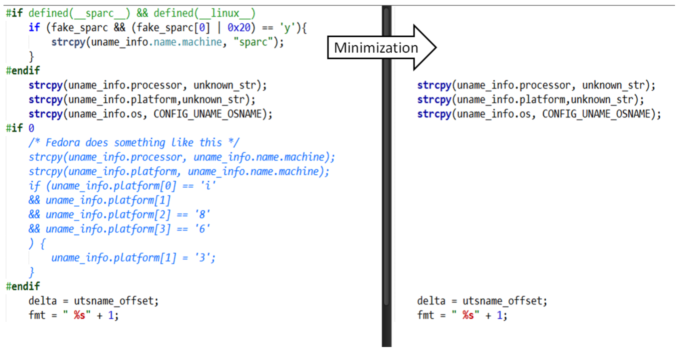
Fig. 2: Minimization technique process flow.

*3) Solution:* As depicted in figure 2, the MakeFile is inherited and CHECK option is tweaked, where existing CHECK feature in kernel MakeFile is replaced with `minimize.py` script, which processes the minimization on the fly with single execution pass. In `make` process, `minimize.py` receives options that are completely similar as the compile flag of each source file along with `$CHECKFLAGS` variable. Below snippet shows on the fly approach:

```
$ make C=1 CHECK=minimize.py
CF="-mindir ../minimized-tree/"
```

The pre-process tweaks the source files with the gcc options `gcc -E -fdirectives-only`. This command allows removal of `#ifdef`, followed by expansion of `#include` but preserving `#define` macros.

The `preprocess()` function available in minimization technique, takes gcc options that are passed via `Makefile` as inputs, which then appends `gcc -E -fdirectives-only` flags and performs preprocess for target C files.
Next is identification and deletion of the expanded header contents that is present in used compiler commands once `make` command is executed. To remove header content, line-markers are used as clues that exists in the preprocessed file of kernel source. For example: `#30 "/usr/include/sys/stsname.h" 2`.
The `stripHeaders()` function in minimization script acquires the preprocessed C file and then search for preprocessor output which is relevant to `#include` lines and is accompanied by deletion of `#include` contents guided by line-markers. `#include` content file name and line number information is conveyed in preprocessor output, for example: In following syntax `#30 "/usr/include/sys/stsname.h" 2`, 30 signifies that this line originates in line 30 of file `utsname.h` after having included in another file which is signified by flag 2. The flag which in this example is indicated by 2 represents returning to the file. However flag 1 signifies start of the file.
This `stripHeader()` algorithm finds the line-markers that starts with `# number` file name and if file name is the target C file then it copies the line and searches for flag. If flag in the line marker is 2 the algorithm marks it `"TO BE REPLACED"` which illustrates "there is `#include` line". Finally the `#include` sentences are restored from the original source code by copying relevant `#include` lines.
The `restoreHeaderInclude()` function in minimization technique carry out header-stripped preprocessed files and searches for `"TO BE REPLACED"` mark, followed by comparing with the original C file and copy original `#include` lines. Once the above steps are accomplished the diff result is only deletion of the unused code without changing `#include` and `#define` lines.



```
#if defined(__sparc__) && defined(__linux__)
    if (fake_sparc && (fake_sparc[0] | 0x20) == 'y'){
        strcpy(uname_info.name.machine, "sparc");
    }
#endif
    strcpy(uname_info.processor, unknown_str);
    strcpy(uname_info.platform,unknown_str);
    strcpy(uname_info.os, CONFIG_UNAME_OSNAME);
#if 0
    /* Fedora does something like this */
    strcpy(uname_info.processor, uname_info.name.machine);
    strcpy(uname_info.platform, uname_info.name.machine);
    if (uname_info.platform[0] == 'i'
    && uname_info.platform[1]
    && uname_info.platform[2] == '8'
    && uname_info.platform[3] == '6'
    ) {
        uname_info.platform[1] = '3';
    }
#endif
    delta = utsname_offset;
    fmt = " %s" + 1;
```

```
    strcpy(uname_info.processor, unknown_str);
    strcpy(uname_info.platform,unknown_str);
    strcpy(uname_info.os, CONFIG_UNAME_OSNAME);
    delta = utsname_offset;
    fmt = " %s" + 1;
```

Fig. 3: Code reduction through minimization technique.

Figure 3 illustrates the minimized code after the `make` process where the `#ifdef` and `#if` blocks are removed. The minimization script `minimize.py` does not support minimization of include files. The main motive behind this exclusion is that, a single include file is referred from multiple C files and resulting minimized include file is not identical for all C files referring the include file. Consequently, if compile option for each C file differ, effective definitions at compile time shall differ too and this differentiate `#ifdef` blocks in the included file.

### B. *Minimization methodology*

*1) Prerequisites:* The script which is developed to exercise minimization approach requires following commands executing in the host machine:

- `diffstat`
- `diff`
- `echo`
- `file`
- `gcc` (Other options that are required to build Linux Kernel or BusyBox)
- `python` (2.x and 3.x compatibility is supported by minimization technique)

*2) Usage:* Proposed minimization technique needs following points for execution of script:

1) Navigate to source directory. Example:
   `$ cd linux-4.4.9`
2) Copy `minimize.py` to kernel directory.
3) Prepare configuration file by tuning the `.config` file and storing it in kernel tree directory. The `.config` can also be generated by executing `make` command. Example:
   `$ make allnoconfig`
4) Add the script directory path. For example:
   `$ export PATH=$PATH:`pwd``
5) Execute `make` with the following `CHECK` options:
   `$ make C=1 CHECK=minimize.py CF="-mindir ../minimized-tree/"`
   Parameter value `C=1` signifies minimization only for (re)compilation target files. `C=2` is used to perform minimization for all the source files regardless of whether they are compilation target or not. Similarly to specify output directory `-mindir` option in `CF` flag is used.

On other hand minimization is also applicable for sub target sources. For example: `$ make drivers C=1 CHECK=minimize.py CF="-mindir ../minimized-tree/"`.
In addition, the script has been modified in such a way that on successful execution, compilation and minimization will be performed at the same time and minimized source tree will be generated under directory `../minimized-tree/`. One thing that needs to be known is that only the target C source files will be minimized. The other file contents(included header etc) remain as they are.

## V. Results

Minimization technique [12] has been experimented and evaluated on platforms such as Linux kernel and BusyBox Tree. The experiment is basically conducted to check reduction metrics after executing minimization technique on original code base.

The evaluation of minimization technique has been implemented on hardware specifications: Processor: 3600MHz, width-64bits, cores-8. Memory: size-7891MiB. Architecture: x86_64.

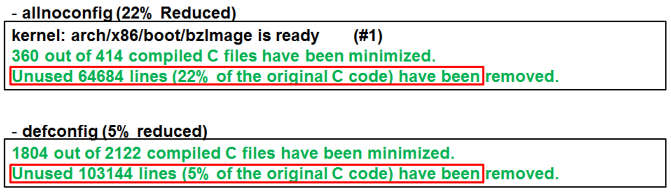
It has been performed by comparing different configurations of target source, particularly `"allnoconfig"` and `"defconfig"`. Main motivation for using different configuration is to comply minimization with expectations, as follows:

- In case of `"allnoconfig"` most features are disabled. This signifies substantial amount of disabled `#ifdef` causing large amount of code reduction. Eventually, it leads to higher minimization ratio.
- Similarly, in case of `"defconfig"`, only a part of features are disabled which leads to less number of disabled `#ifdef` resulting in less amount of code reduction. Hence in case of `"defconfig"` reduction is expected to be lower than `"allnoconfig"`.

### A. *Linux Kernel*

Implementation on Linux kernel with `"allnoconfig"` and `"defconfig"` option results in substantial reduction of unnecessary code has been achieved as shown in figure 4. The metrics are as follows:

- `allnoconfig`: 64684 unused lines were removed from kernel source which constitutes around 22% of original C code in kernel source.
- `defconfig`: With this option 103144 unused lines were removed from kernel source that comprises about 5% of original C code.



```
- allnoconfig (22%  Reduced)
kernel: arch/x86/boot/bzImage is ready      (#1)
360 out of 414 compiled C files have been minimized.
Unused 64684 lines (22%  of the original C code) have been removed.

- defconfig (5%  reduced)
1804 out of 2122 compiled C files have been minimized.
Unused 103144 lines (5%  of the original C code) have been removed.
```

Fig. 4: Minimization technique execution on Linux Kernel.

The minimization script `minimize.py` executes not only for limited configurations, but also other customized ones including PREEMPT_RT patch.

### B. *BusyBox Tree*

On executing minimization technique in BusyBox tree having `"allnoconfig"` and `"defconfig"` configuration options, the reduction metrics obtained are as follows:

- `allnoconfig`: 51 out of 112 compiled C files have been minimized. 5945 lines (34% of original C code) unused lines were removed.
- `defconfig`: 296 out of 505 compiled C files have been minimized. 20453 lines (11% of original C code) unused lines were removed.

### C. *Quantification of other OSS projects*

Apart from Linux Kernel and BusyBox tree, quantification of `#ifdef` and `#if-blocks` that could potentially be removed from open-source project ARCTIC Core source code [13] as compared to Linux Kernel has been exercised.
The motive is to quantify how much beneficial can Minimization approach be for OSS projects such as ARCTIC Core. The quantification is carried out by finding total number of `#ifdef` and `#if-block` and calculating the ratio with total lines of code as below:

| Complexity Metrics | Linux Kernel | | | BusyBox Tree | | | PREEMPT_RT | |
|---|---|---|---|---|---|---|---|---|
| | Original Source | Minimized(x86_defconfig) | Minimized(allnoconfig) | Original Source | Minimized(x86_defconfig) | Minimized(allnoconfig) | Original | Minimized |
| Average Line Score | 23 | 7 | 5 | 22 | 21 | 19 | 10 | 7 |
| 50%-ile score | 4 | 3 | 2 | 9 | 9 | 5 | 4 | 3 |
| Highest Score | 1846 | 194 | 158 | 283 | 283 | 283 | 530 | 194 |

TABLE I: Complexity metrics in original and minimized targets.

```
Total number of lines in all C files
of Arctic Core source code = 407994 lines.
Total number of #ifdef existing = 12744.
Number of lines that can
be reduced = 12744/407994*100 = 3.12%
```

Similarly, in Linux Kernel,

```
Total number of lines in all
C files = 15086494 lines.
Total number of #ifdef existing = 85728.
Number of lines that can be reduced =
85728/15086494*100 = 0.568%
```

The statistics above indicates that there are more (approximately 5.5 times higher) chances in Arctic Core of eliminating unused `#ifdef` switches. This can be stated as a possible advantage of Minimization technique, however port implementation is yet to be realised.

## VI. Evaluation

### A. Complexity statistics

To analyze the complexity of ”C” program function, Linux with PREEMPT_RT patch, Linux Kernel source and BusyBox tree has been evaluated by comparing complexities of C program functions of minimized and original source code of these targets respectively. The statistics have been acquired using ”Complexity” tool [14].
The complexity tool has been used because it helps extensively in getting an idea of how much effort may be required to understand and maintain the code. Higher the score, more complex is the procedure, and minimization shows comparably lower complexity score which signifies it is easy to read and maintain [14].

Table I illustrates the measured complexities of original and minimized targets (Linux kernel, BusyBox tree and PREEMPT_RT Kernel) respectively. For Linux kernel and BusyBox `allnoconfig` and `x86_defconfig` configurations has been evaluated for minimized code. The minimized code demonstrate decreased complexity in terms of average line score, 50%-ile score and highest score in all three targets.

### B. Verification for the minimized built binary

The disassembled code (”objdump -d”) matches the binaries that are built from minimized and original source code. Also the configuration and target has been confirmed based on Busybox and Linux kernel as below:

- BusyBox-1.24.1: Checked configuration options include `defconfig` and `allnoconfig`.
- Linux kernel-4.4.1: Configuration options verified `allnoconfig`.

## VII. Benefits

### A. Verification time and cost improvement

For verification time improvement static analysis has been implemented by comparing results of original and minimized kernel source tree using Coccinelle which is a program matching and transformation engine for C code and has many semantic patches to the new submissions to the mainline kernel repository [15], [16]. The verification has been implemented by executing a semantic patch [2] which detects functions whose declared return value type and actually returned type differs by scanning source files (*.c and *.h) that are referred from `init/main.c` in kernel tree. Results of the static verification in terms of time parameter are illustrated below:
Average spatch execution time:

```
Original Kernel Source: 12.37[s]
Minimized Kernel Source: 2.24[s]
```

The minimized technique provide around 5.5 times faster analysis as compared to original kernel source tree.

### B. False Positive reduction

False positive is a test result which wrongly indicates that a particular condition or attribute is present. To mitigate such situation static analysis [15] was conducted on original and minimized kernel source tree. The number of meaningless detection were mitigated as follows in the minimized kernel source. Number of detection using Coccinelle:

```
Original Kernel Source: 126
Minimized Kernel Source: 82
```

### C. Easy Code Inspection

The minimization technique generates easy to read source code by implementing following assimilation:

- Unused `#ifdef`, `#if` blocks are removed.
- `#include` and `#define` lines are preserved.
- Producing same binary file as that of original source tree.

### D. Pruning function call graph:

During analysis, it is required to identify every possible call path to establish and trace relationship between program and subroutines, callgraph is a directed graph that represents this relationship [17]. The call graph displays every function call regardless of `#ifdef` switches which results in substantially complex graph which is difficult to trace. With minimization technique, call graph display illustrates only used function calls

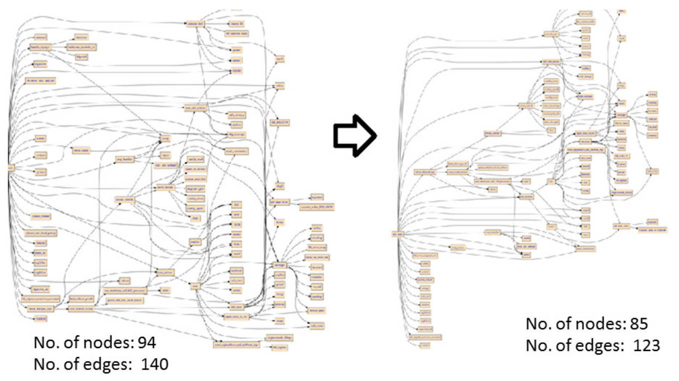


Fig. 5: Call graph for Linux kernel before (left) and after (right) minimization.

thereby providing minimized search space. Figure 5 illustrates call graph transformation before and after minimization.

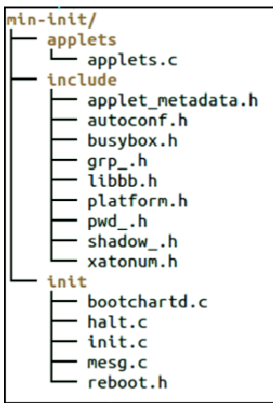
With minimization the number of nodes reduced from 94 to 85 followed by edges which are from 140 to 123 hence a narrow search space.

### *E. Extracting minimal subtarget sources:*

To easily identify which files are used in source tree for efficient software walk-through, subtarget can be specified in the minimized command in result of which minimization will extract only the used source files. The following snippet shows addition of subtarget *init* in minimized command:

```
$ make init C=2 CHECK=minimize.py
CF="-mindir ../min-init"
```

This results in extraction of only used source files when subtarget is defined and is shown in figure 6.



```
min-init/
├── applets
│   └── applets.c
├── include
│   ├── applet_metadata.h
│   ├── autoconf.h
│   ├── busybox.h
│   ├── grp_.h
│   ├── libbb.h
│   ├── platform.h
│   ├── pwd_.h
│   ├── shadow_.h
│   └── xatonum.h
└── init
    ├── bootchartd.c
    ├── halt.c
    ├── init.c
    ├── mesg.c
    └── reboot.h
```

Fig. 6: Depended *.c files of Linux kernel in minimized form. Actually included *.h files.

## VIII. Conclusion

The minimization technique helps substantially in improving the readability of source code which results in efficient code review and inspection. It helps in narrowing down search space by giving evidence for unused code. The evaluation of this technique has been performed on target platform such as Linux Kernel, BusyBox Tree and PREEMPT_RT Kernel. Minimization reduction of approximately 5% is achieved across the PREEMPT_RT Linux kernel, Linux kernel and the Busybox software. From analysis stand-point, this provide essential benefits such as reduction in verification time (spatch execution) from 12.37[s] in original kernel source to 2.24[s] in minimized kernel, false positive reduction where the number of detection relating to bugs using Coccinelle (static analysis) reduces from 126 to 82.
This helps in application domains such as automotive, railways, industry etc. The future work for minimization technique includes extension to other compilers such as LLVM [18] followed by adaption with architecture such as ARM; various build system e.g. CMake, automake. Binary equivalence is checked, however formal equivalence between the original and minimized source code tree is still a future work. The source code is available at GitHub [12].

## References

[1] E. Andersen. Busybox. [Online]. Available: https://www.busybox.net

[2] W. Sang. Evolutionary development of a semantic patch using coccinelle. [Online]. Available: http://lwn.net/Articles/380835/

[3] S. Zhou, J. Al-Kofahi, T. N. Nguyen, C. Kästner, and S. Nadi, "Extracting configuration knowledge from build files with symbolic analysis," in *Proceedings of the Third International Workshop on Release Engineering*, ser. RELENG '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 20–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=2820690.2820700

[4] GCC-Team. Gcc, the gnu compiler collection. Free Software Foundation, Inc. [Online]. Available: https://gcc.gnu.org/

[5] D. Gilbert. (2003, August) The linux 2.4 scsi subsystem howto. Linux Document Project. [Online]. Available: http://www.tldp.org/HOWTO/SCSI-2.4-HOWTO/kconfig.html

[6] Y. Xuejun, C. Yang, E. Eric, and R. John. Csmith. [Online]. Available: https://embed.cs.utah.edu/csmith/

[7] GNU-Manual, *The C preprocessor*, GNU. [Online]. Available: https://gcc.gnu.org/onlinedocs/cpp/index.html

[8] StackOverflow. Strip linux kernel sources according to .config. [Online]. Available: http://stackoverflow.com/questions/7353640/strip-linux-kernel-sources-according-to-config

[9] S. Poznyakoff. Gnu cflow. GNU. [Online]. Available: http://www.gnu.org/software/cflow/

[10] A. Younis, Y. K. Malaiya, and I. Ray, "Assessing vulnerability exploitability risk using software properties," *Software Quality Journal*, vol. 24, no. 1, pp. 159–202, 2016. [Online]. Available: http://dx.doi.org/10.1007/s11219-015-9274-6

[11] P. J. Salzman. The linux kernel module programming guide. [Online]. Available: http://www.tldp.org/LDP/lkmpg/2.4/html/x208.html

[12] K. Hashimoto, *The Minimization script*. [Online]. Available: https://github.com/Hitachi-India-Pvt-Ltd-RD/minimization

[13] ArcticCore. Arctic core autosar 3.1 repositories. [Online]. Available: www.arccore.com/resources/repositories

[14] B. Korb. Measure complexity of c source. [Online]. Available: https://www.gnu.org/software/complexity/manual/complexity.html

[15] G. Muller. (2015, October) Coccinelle. INRIA; LIP6; IRILL. [Online]. Available: http://coccinelle.lip6.fr/documentation.php

[16] V. Nossum. Impact on the linux kernel. [Online]. Available: http://coccinelle.lip6.fr/impact_linux.php

[17] G. Kaszuba. Python call graph. [Online]. Available: http://pycallgraph.slowchop.com/en/master/guide/intro.html

[18] The llvm compiler infrastructure. [Online]. Available: http://www.llvm.org