Effective Source Code Analysis with Minimization

July 5, 2016

Geet Tapan Telang

Research Engineer – IT Platform
Hitachi India Pvt. Ltd.

Contents

1. Introduction

OSS Demand

- Growing demand for OSS/Linux in Safety Critical domain.
- Size of code is approximately 20 million lines of code (Linux OS).
- Validation and analysis makes traditional methods difficult to follow.
- Code coverage and analysis is major part of verification and validation.
- Scoping the target code is a big challenge.

Problem

“#ifdef disasters”

/drivers/dma/dmaengine.c

- The #ifdefs makes the code hard to:
  - Review
  - Debug
  - Maintain
  - Verify

```c
static bool device_has_all_tx_types(struct dma_device *device)
{
	/* A device that satisfies this test has channels that will never cause
	 * an async_tx channel switch event as all possible operation types can
	 * be handled.
	 */
	#ifdef CONFIG_ASYNC_TX_DMA
	if (!dma_has_cap(DMA_INTERRUPT, device->cap_mask))
		return false;
	#endif

	#if defined(CONFIG_ASYNC_MEMCPY) || defined(CONFIG_ASYNC_MEMCPY_MODULE)
	if (!dma_has_cap(DMA_MEMCPY, device->cap_mask))
		return false;
	#endif

	#if defined(CONFIG_ASYNC_XOR) || defined(CONFIG_ASYNC_XOR_MODULE)
	if (!dma_has_cap(DMA_XOR, device->cap_mask))
		return false;

	#ifndef CONFIG_ASYNC_TX_DISABLE_XOR_VAL_DMA
	if (!dma_has_cap(DMA_XOR_VAL, device->cap_mask))
		return false;
	#endif
	#endif

	#if defined(CONFIG_ASYNC_PQ) || defined(CONFIG_ASYNC_PQ_MODULE)
	if (!dma_has_cap(DMA_PQ, device->cap_mask))
		return false;

	#ifndef CONFIG_ASYNC_TX_DISABLE_PQ_VAL_DMA
	if (!dma_has_cap(DMA_PQ_VAL, device->cap_mask))
		return false;
	#endif
	#endif

	return true;
}
```

“#ifdef disasters”

HITACHI
Inspire the Next

/drivers/dma/dmaengine.c

```
static bool device_has_all_tx_types(struct dma_device *device)
{
        /* A device that satisfies this test has channels that will never cause
         * an async_tx channel switch event as all possible operation types can
         * be handled.
         */
        if (!dma_has_cap(DMA_INTERRUPT, device->cap_mask))
                return false;

        if (!dma_has_cap(DMA_PQ, device->cap_mask))
                return false;

        return true;
}
```

If code is free from #ifdef blocks then, analysis shall be more effective.

Is there a way ?

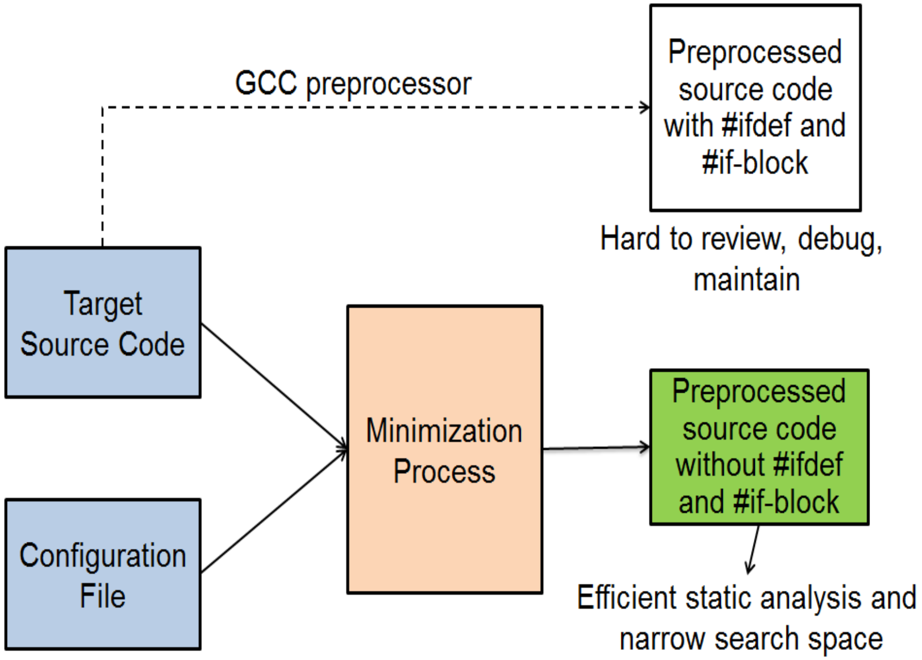
Approach

The Minimization Approach

- The minimization approach tweaks integrated MakeFile options to produce compilable stripped code.

- Signifies efficient way to get a set of stripped kernel source code based on a .config file.

- Generate source tree where;
  - Unused #ifdef, #if blocks have been removed
  - #include and #define lines are preserved
  - Only used source files exist
  - Produces the same binary file as the original tree



The Minimization Approach

This code transformation is what we term as Minimization.

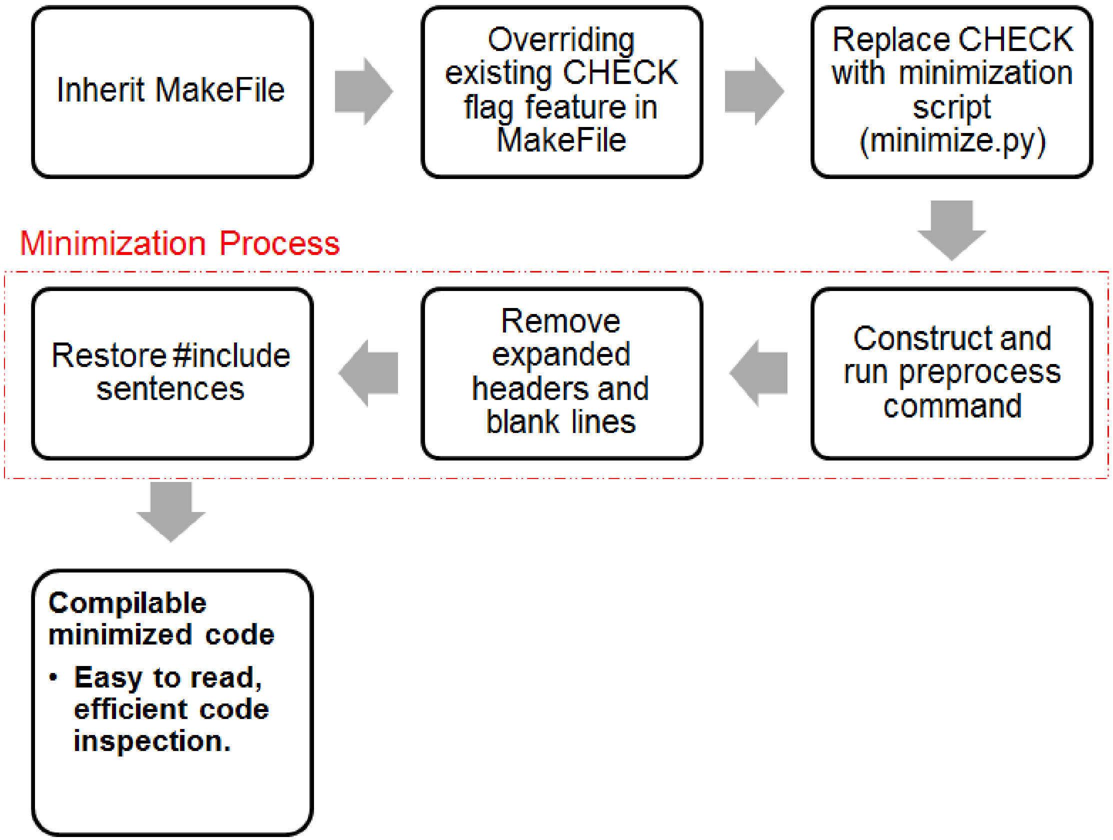
Original idea of using GREP (Approach-I)

- Requires complete build in advance.
- Text parsing has to be acquired from build log.
- Source code modification to remove redundant code.

Too much user Involvement!!!

Minimize.py script (Approach-II)

- MakeFile integration
  - Override existing CHECK flag feature
- Minimizing procedure
  - Preprocess, expanded header restoration
- Binary verification
  - Compare “minimized binary” and the original



MakeFile Integration

- Override existing CHECK feature in kernel MakeFile

```
kotaro@kotaro-OptiPlex-7020:~/Minimization/linux-4.3.3$ make help | grep CHECK
  make C=1   [targets] Check all c source with $CHECK (sparse by default)
  make C=2   [targets] Force check of all c source with $CHECK
```

- Makefile of the root directory:

```
CHECK           = sparse

CHECKFLAGS     := -D__linux__ -Dlinux -D__STDC__ -Dunix -D__unix__ \
                  -Wbitwise -Wno-return-void $(CF)
```

- Minimization script(minimize.py) usage:

  Replace CHECK with minimize.py so make can process minimization

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

In make process, "minimize.py" will receive the same option as the compile flags of each source file, plus $CHECKFLAGS variable.

ON THE FLY GENERATION (no post processing)!!!

Minimization procedure

HITACHI
Inspire the Next

1. Preprocess the source files

   gcc –E –fdirectives-only

   ⇨ #ifdef block disappears, #include gets expanded,
   but #define macros are preserved, also removes empty lines

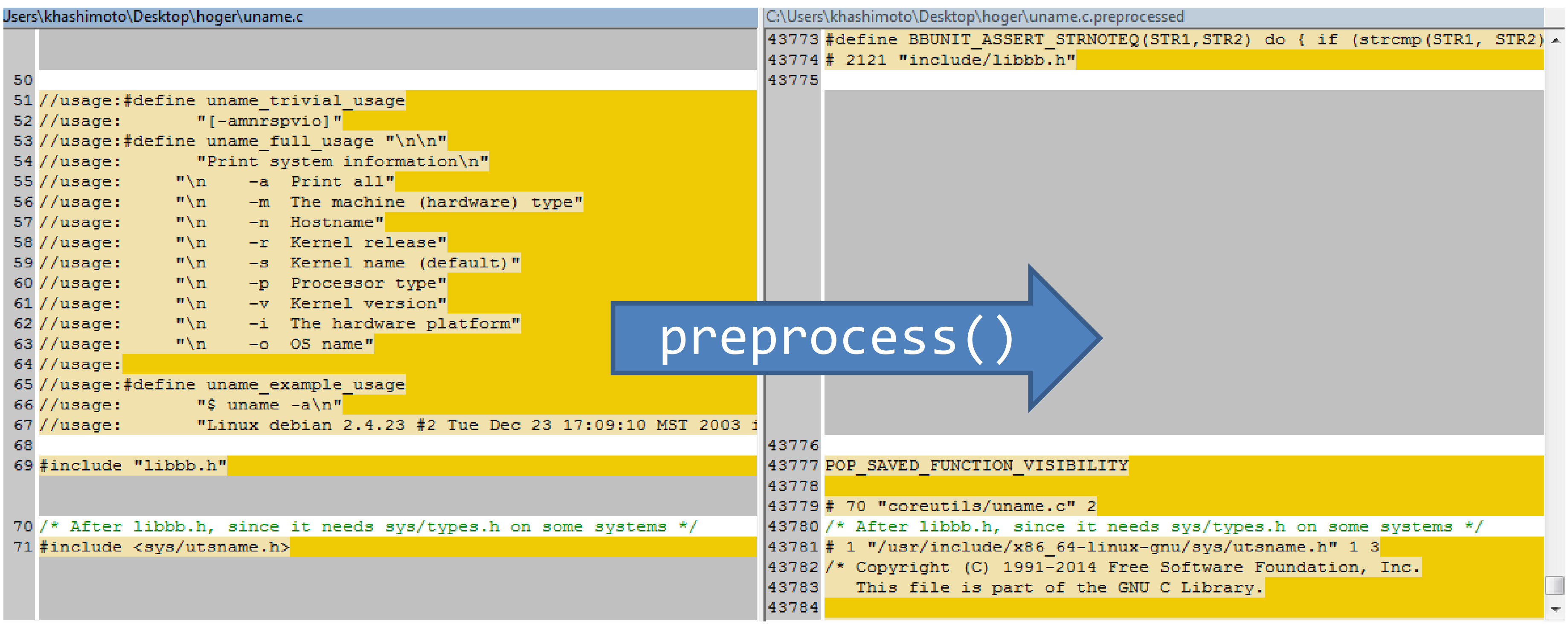
2. Identify & delete the expanded header contents
   - Use clues(linemarkers) that exist in the preprocessed file
   - Example of linemarkers: # 30 “/usr/include/sys/stsname.h” 2

3. Restore #include sentences
   - Copy relevant #include lines from the original source

Preprocess the source file

HITACHI
Inspire the Next

- `preprocess()` function in minimize.py
  - Takes gcc options passed via Makefile
  - Appends “-E –fdirectives-only” flags
  - Perform preprocess for the target C file



Identify & delete the expanded headers

HITACHI
Inspire the Next

- `stripHeaders()` function in minimize.py
  - Takes preprocessed C file
  - Search Preprocessor Output relevant to #include lines
  - Delete included contents guided by the *linemarkers*

Included file name and line number information is conveyed in the preprocessor output; *linemarkers*

LineMarker Ex. `# 30 “/usr/include/sys/utsname.h” 2`

linenum

filename

flags

It means, the following lines originated in line 30 of utsname.h, after having included another file(flag:2).

https://gcc.gnu.org/onlinedocs/cpp/Preprocessor-Output.html

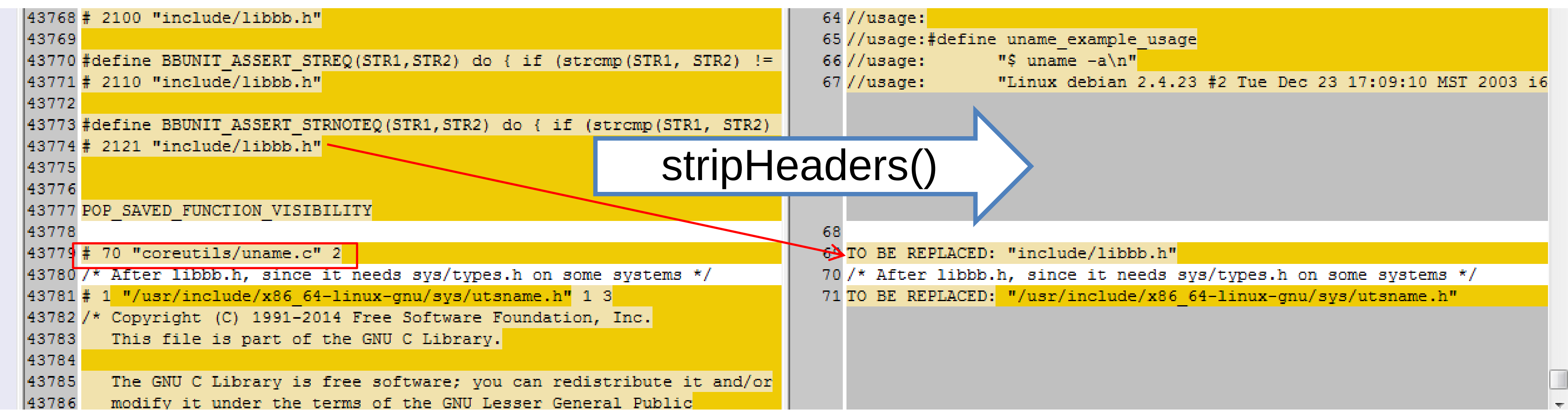
Flags:
  1: indicates the start of the new file
  2: indicates returning to the file.

Identify & delete the expanded headers

HITACHI
Inspire the Next

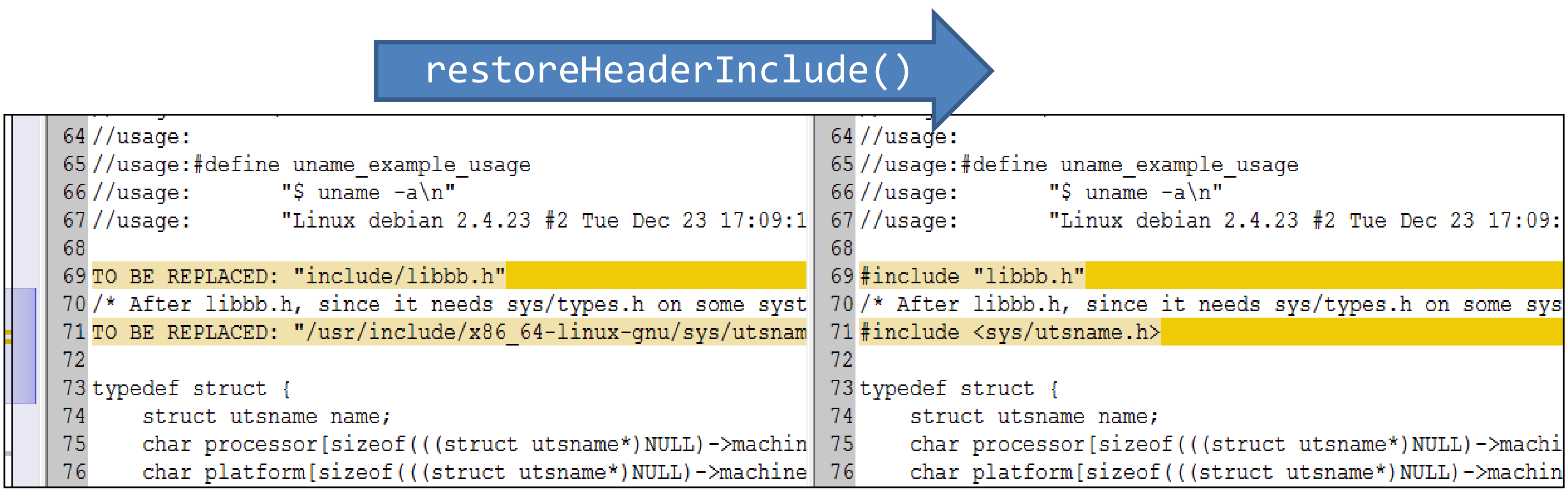
- stripHeaders() algorithm
  - Find linemakers (starting with '# *number* "*filename*"')
  - If *filename* is the target C file:
    - copy the following lines
    - And if *flag* in the linemaker is 2:
      - Mark "TO BE REPLACED" that means "there is #include line"

Flag 2 indicates returning to the file (after having included another file).



Restore #include sentences

- restoreHeaderInclude() function in minimize.py
  - Takes header-stripped preprocessed file
  - Look for “TO BE REPLACED” marks
  - Compare with the original C file, copy original #include lines



Minimization Diff

HITACHI
Inspire the Next

- Finally, diff result is only deletions of the unused code.
  - Without changing #include, #define lines.
  - Minimization also removes blank lines which comprised of unused code.

```c
#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;
```

Minimization

```c
    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;
```

2. Results

Minimization Results

Linux Kernel Tree

- allnoconfig: 64684 unused lines were removed → 22% of original C code.
- defconfig: 103144 unused lines were removed → 5% of original C code.

BusyBox Tree

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

ARCTIC Core source code

- Statistics shows approximately 5.5 times higher chances of eliminating unused #ifdef switches.

Evaluation

Minimization Evaluation

HITACHI
Inspire the Next

Complexity Statistics

- To analyze the complexity of "C" program function.
- Linux with PREEMPT_RT patch, Linux Kernel source, BusyBox tree as shown in table below.
- Complexity (a GNU utility) tool has been used.

Disassembled code("objdump –d") matches

- Between the binaries built from minimized source and original one.
- Confirmed configuration & target:
  - BusyBox-1.24.1: defconfig, allnoconfig
    - busybox (executable)
  - Linux kernel 4.4.1: allnoconfig
    - vmlinux.o

Minimized code is compilable and produces same binary

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

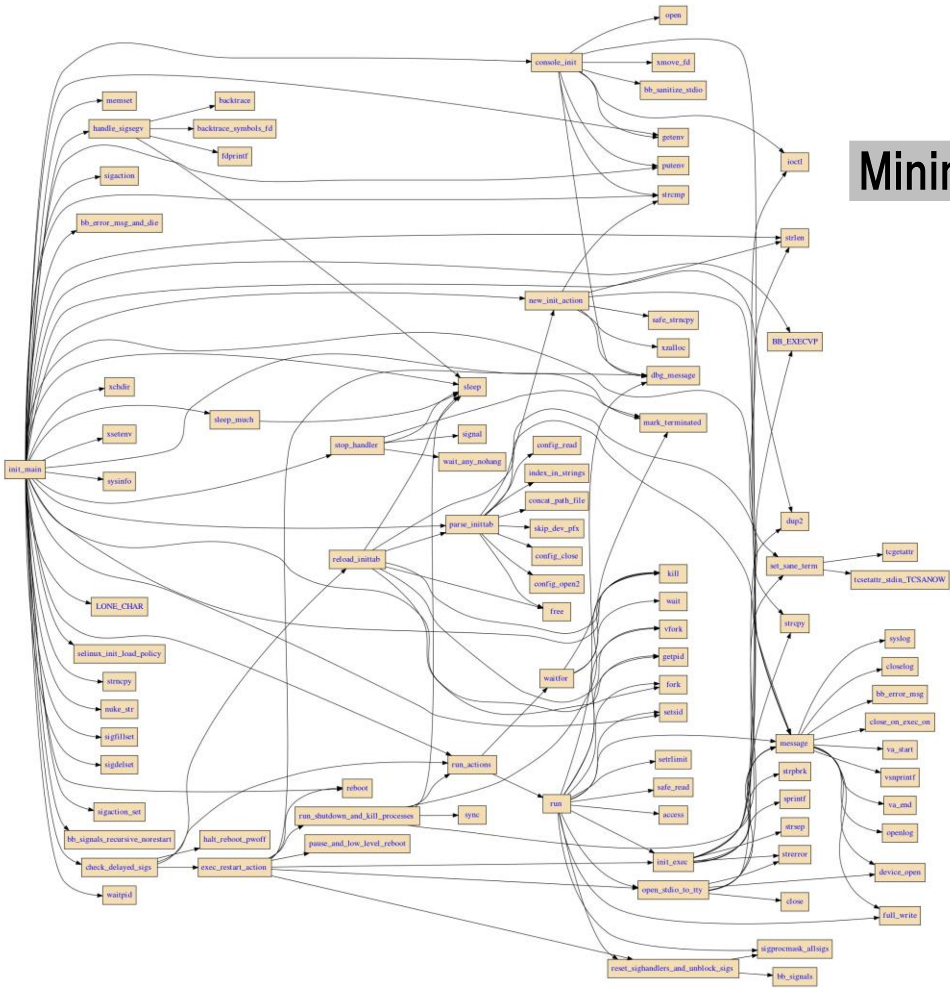
Measured complexity in terms of average line score, 50%-ile score and highest score.

Benefits

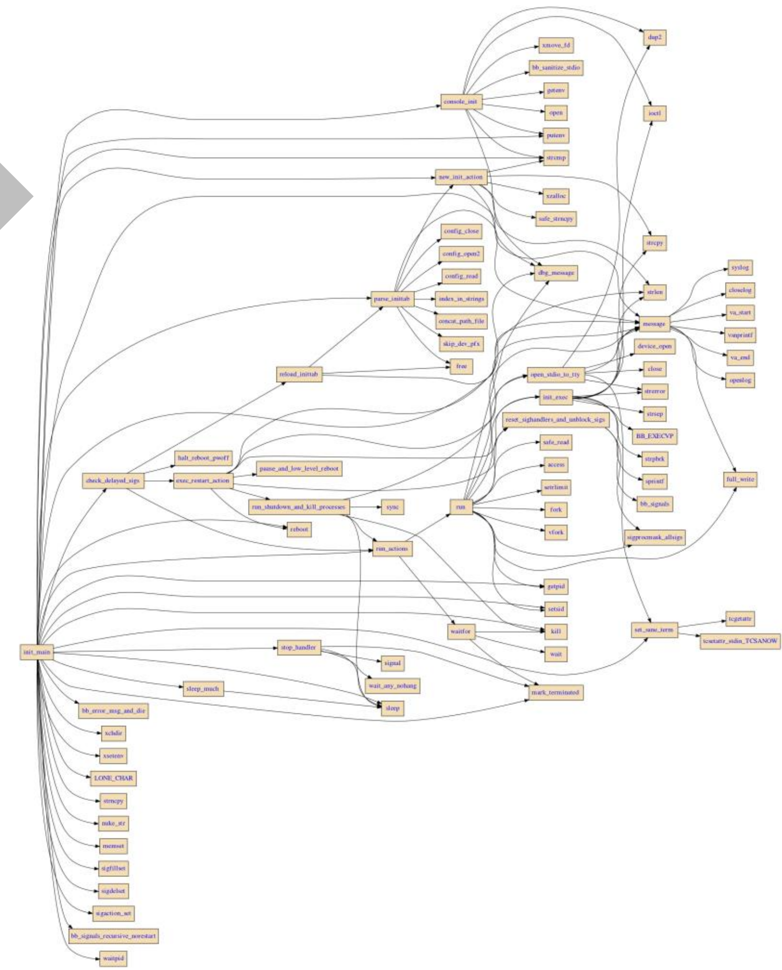
Benefits

- Verification time and cost improvement
  - Static analysis through Coccinelle
  - Executed a semantic patch for detecting functions have different return type values
  - Statistics
    - Comparison of execution time and minimization was faster.
    - 12[s] and 2.24[s] for original and minimized kernel respectively.
- False positive reduction
  - Wrong indication about presence of particular condition.
  - Statistics
    - Original kernel source: 126
    - Minimized kernel source: 82
- Pruning function call graph
  - Analysis requires every possible call path to establish and trace relationship between program and subroutines.
  - Call graph is a directed graph that represents this relationship.

Benefits



Minimization



No. of nodes: 94
No. of edges: 140

No. of nodes: 85
No. of edges: 123

Benefits

Extracting Minimal Subtarget Sources

```
$ cd busybox-1.24.1
$ make init C=2 CHECK=minimize.py CF="-mindir ../min-init"
```

If subtarget is specified in the minimized command,
Only the used source files will be extracted.

```
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
```

Depended *.c files in minimized form.
Actually included *.h files

- Easy to identify which files are used
- Helps efficient software walk-through

3. Conclusion

Conclusion

- Improves readability for human.
  - Helps efficient code review / inspection.
- Narrows down “search space”.
  - Gives evidence for unused code.
  - Saves verification cost (time & space).
  - Achieves higher test coverage.
  - Reduces false-positives.
- From analysis stand-point, this provides
  - Reduction in verification time
  - False-positive reduction
- Much more potential for domains like safety and mission critical systems.

Future Work

HITACHI
Inspire the Next

- To adapt more config / architecture
  - More than allnoconfig, defconfig / x86, arm
- To adapt more projects
  - For different build system (automake, CMake etc.)
- To prove minimized tree is “equal” to original one
  - How to formally verify equivalence???
- To find out more applications
  - Something that enhances existing tools / techniques
- Available in:
  - https://github.com/Hitachi-India-Pvt-Ltd-RD/minimization

END

Effective Source Code Analysis with Minimization

July 5, 2016

Geet Tapan Telang

Hitachi India Pvt. Ltd.