Specialization of Run-time Configuration Space at Compile-time: An Exploratory Study

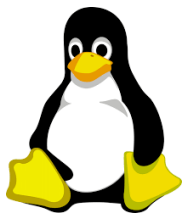
Xhevahire Tërnava[1], Mathieu Acher[12], Benoit Combemale[1]

[1]Université de Rennes 1, CNRS, Inria, IRISA
[2]Institut Universitaire de France (IUF)
Rennes, France

SAC SE 2023, March 27 - 31, 2023

Context

- Today's software systems are highly-configurable, *e.g.*,
  - Linux: ≈20K options; $\approx 2^{20,000}$ configuration space (?)
  - Renault: $\approx 10^{22}$ cars (?) - 5 million light-years [1]
  - x264: 39 compile-time options and 162 run-time options

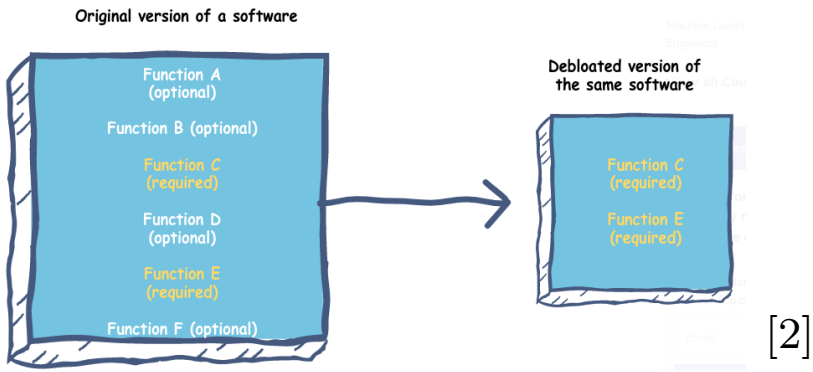



```
[xternava@slimfast (Fedora 35) coreutils9.1]$ ./src/cat --help
Usage: ./src/cat [OPTION]... [FILE]...
Concatenate FILE(s) to standard output.

With no FILE, or when FILE is -, read standard input.

  -A, --show-all           equivalent to -vET
  -b, --number-nonblank    number nonempty output lines, overrides -n
  -e                       equivalent to -vE
  -E, --show-ends          display $ at end of each line
  -n, --number             number all output lines
  -s, --squeeze-blank      suppress repeated empty output lines
  -t                       equivalent to -vT
  -T, --show-tabs          display TAB characters as ^I
  -u                       (ignored)
  -v, --show-nonprinting   use ^ and M- notation, except for LFD and TAB
      --help        display this help and exit
      --version     output version information and exit
```

1: Renault Group, Keynote Yves Bossu at the 17th International Working Conference VaMoS. 2023

Context

- Up to 54.1% of options are rarely set by any user [1]
  - $\rightarrow$ *i.e.*, they bloat the software
- Software debloating is the process of eliminating superfluous functionalities from it, with the goal of reducing its execution time, resource consumption, and attack surface

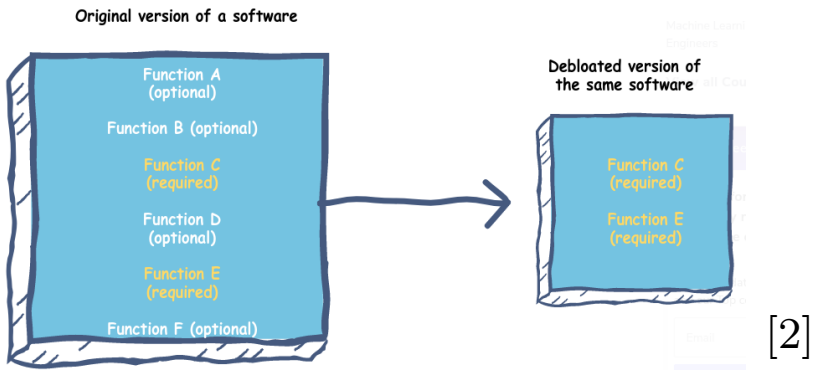


[2]

1: Xu T. et al. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In the 10th Joint Meeting on Foundations of SE. 2015

Context

- Up to 54.1% of options are rarely set by any user [1]
  - $\rightarrow$ *i.e.*, they bloat the software
- Software debloating is the process of eliminating superfluous functionalities from it, with the goal of reducing its execution time, resource consumption, and attack surface



[2]

- Debloating source code, compiled binaries, libs, or flags
- Idea: specializing the run-time configuration space of a system

---

1: Xu T. et al. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In the 10th Joint Meeting on Foundations of SE. 2015

2: Source: https://www.educative.io/edpresso/what-is-software-debloating

x264: motivation case study

Most of the C-based systems are configurable through the compile-time and rune-time options. For example, the system of **x264** video encoder

x264: db0d417 commit; 114,475 LoC; 2

x264's options: 39 compile-time; 162 run-time

E.g., x264 build without mp4 support

```
$ ./configure --disable-lsmash && make
```

E.g., video encoding with x264 using three run-time options

```
$ x264 --no-cabac --mbtree --mixed-refs -o vid.264 vid.y4m
```

x264 has 10 presets, but they contain only 22 options, *i.e.*, 140 unchanged

Problem statement

Proposal: debloating run-time options at compile time

In a given context, some run-time options may never be set
e.g., `--no-cabac` in x264

- Likely, they increase the system's binary size,
- they augment the system's attack surface (# gadgets),
- they may slow down the system, ...

Removing unused run-time variability in a software system is not trivial

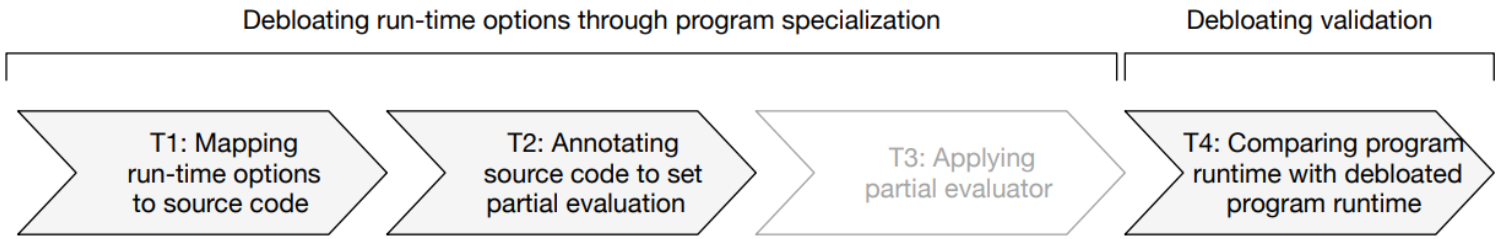
Vision:

- Developers should have the means to remove unused run-time options
- And, to possibly build new x264's variants

  e.g., x264-hq, x264-fast, x264-tinyfast, x264-secured, etc.

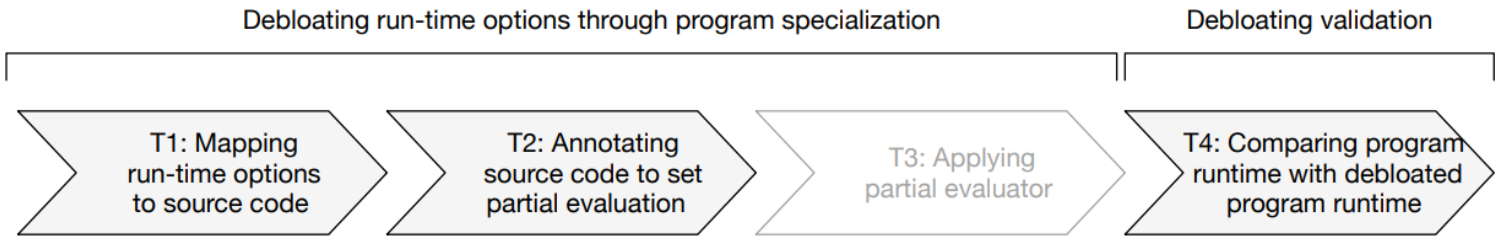
Our specialization approach

Debloating process



Our specialization approach

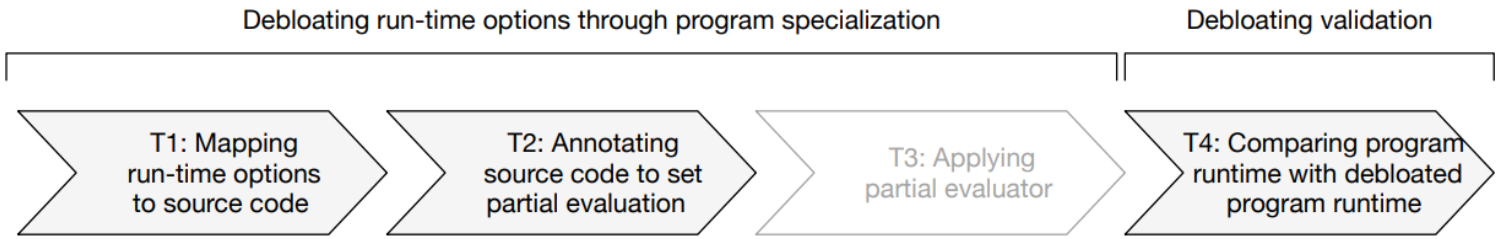
Debloating process



T1: Implementation patterns (getopt.h, a variable per option, etc.)

Our specialization approach

Debloating process

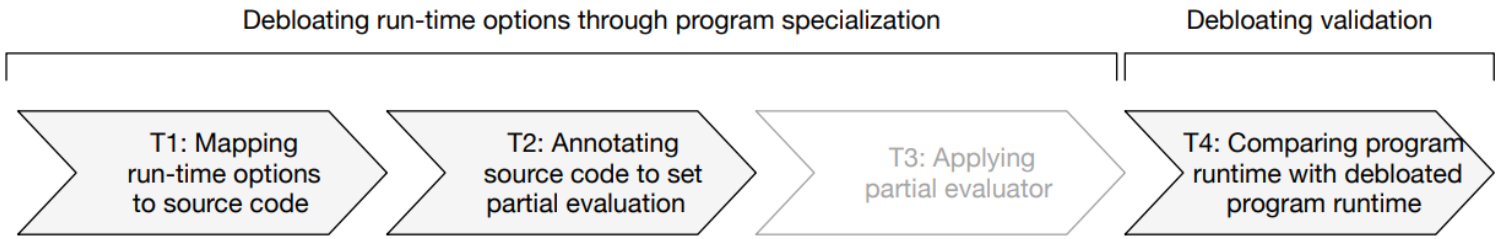


T1: Implementation patterns (getopt.h, a variable per option, etc.)

T2: Options are made explicit using C preprocessor directives

Our specialization approach

Debloating process



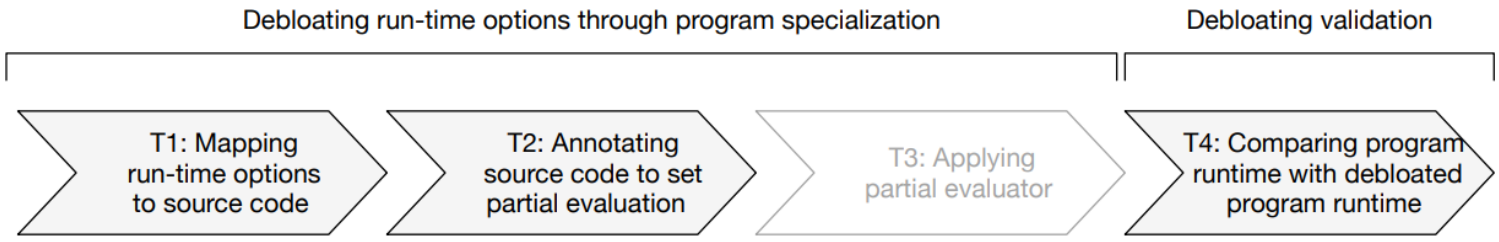
T1: Implementation patterns (getopt.h, a variable per option, etc.)

T2: Options are made explicit using C preprocessor directives

```
1  /* File removeoption.h */
2  #ifndef CABAC_YES
3  #define CABAC_YES 0
4  #endif
5  #ifndef CABAC_NO
6  #define CABAC_NO 1
7  #endif
8  /* The rest of the directives are omitted */
```

Our specialization approach

Debloating process



T1: Implementation patterns (getopt.h, a variable per option, etc.)

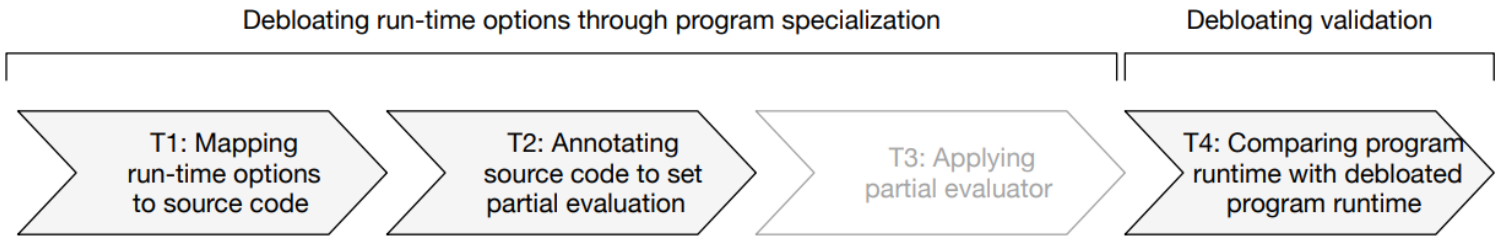
T2: Options are made explicit using C preprocessor directives

```
1  /* File encoder/encoder.c */
2  /*The previous code is omitted*/
3      if( h->param.b_cabac )
4          x264_cabac_init( h ); //option --caba
5      else
6          x264_cavlc_init( h ); //option --no-c
7  /*The rest of the code is omitted*/
```

```
1  /* File encoder/encoder.c */
2  /*The previous code is omitted*/
3  #if CABAC_YES
4      if( h->param.b_cabac )
5          x264_cabac_init( h ); //option --cabac
6  #endif
7  #if CABAC_YES && CABAC_NO
8      else
9  #endif
10 #if CABAC_NO
11         x264_cavlc_init( h ); //option --no-cabac
12 #endif
13 /*The rest of the code is omitted*/
```

Our specialization approach

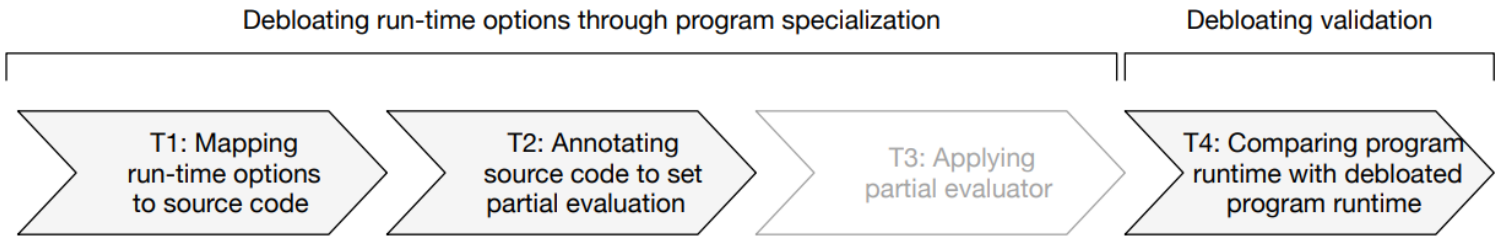
Debloating process



- T1: Implementation patterns (getopt.h, a variable per option, etc.)
- T2: Options are made explicit using C preprocessor directives
- T3: Preprocessor switches

Our specialization approach

Debloating process



- T1: Implementation patterns (getopt.h, a variable per option, etc.)
- T2: Options are made explicit using C preprocessor directives
- T3: Preprocessor switches
- T4: The specialized system should be compilable and valid (video size, quality, and system interoperability)
  - If not T4, we then repeat T1 to T4

Experimental design

Null and alternative hypothesis

H_{01} : The baseline software system and its specialization are not significantly different with respect to their binary size

H_{A1} : The specialized software system has smaller binary size than its baseline

H_{02} : The baseline software system and its specialization are not significantly different with respect to their attack surface

H_{A2} : The specialized software system has smaller attack surface than its baseline

H_{03} : The baseline software system and its specialization do not perform significantly different

H_{A3} : The specialized software system performs better than its baseline as for the encoding time, bitrate, and frame rate

Experimental design

Sample of 20 specializations

- Baseline system (S_0): x264 with default configuration in db0d417
- 20 specialized systems: 10 run-time options and 10 presets

Options \ Presets		ultrafast	superfast	veryfast	faster	fast	medium	slow	slower	veryslow	placebo
		S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	S_{17}	S_{18}	S_{19}	S_{20}
--no-mixed-refs	S_1	●	●	●	●	○	○	○	○	○	○
--no-mbtree	S_2	●	●	○	○	○	○	○	○	○	○
--no-cabac	S_3	●	○	○	○	○	○	○	○	○	○
--no-weightb	S_4	●	○	○	○	○	○	○	○	○	○
--no-psy	S_5	○	○	○	○	○	○	○	○	○	○
--mixed-refs	S_6	○	○	○	○	●	●	●	●	●	●
--mbtree	S_7	○	○	●	●	●	●	●	●	●	●
--cabac	S_8	○	●	●	●	●	●	●	●	●	●
--weightb	S_9	○	●	●	●	●	●	●	●	●	●
--psy	S_{10}	●	●	●	●	●	●	●	●	●	●

○ - unused option; ● - used option

Scenario$_1$: $S_1 - S_{10}$ system's specializations by a single option

Scenario$_2$: $S_{11} - S_{20}$ system's specializations regarding a preset

Experimental design

The inputs and workstation used

Inputs: 8 video, representative of 1300 videos in YouTube UCG dataset

Name [.mkv]	Size [MB]	Length [sec.]	Name [.mkv]	Size [MB]	Length [sec.]
V_1_720x480	108.4	13	V_5_640x360	172.8	20
V_2_480x360	155.6	20	V_6_640x360	41.5	20
V_3_640x360	165.5	19	V_7_640x360	207.4	20
V_4_640x360	172.5	19	V_8_624x464	217.2	20

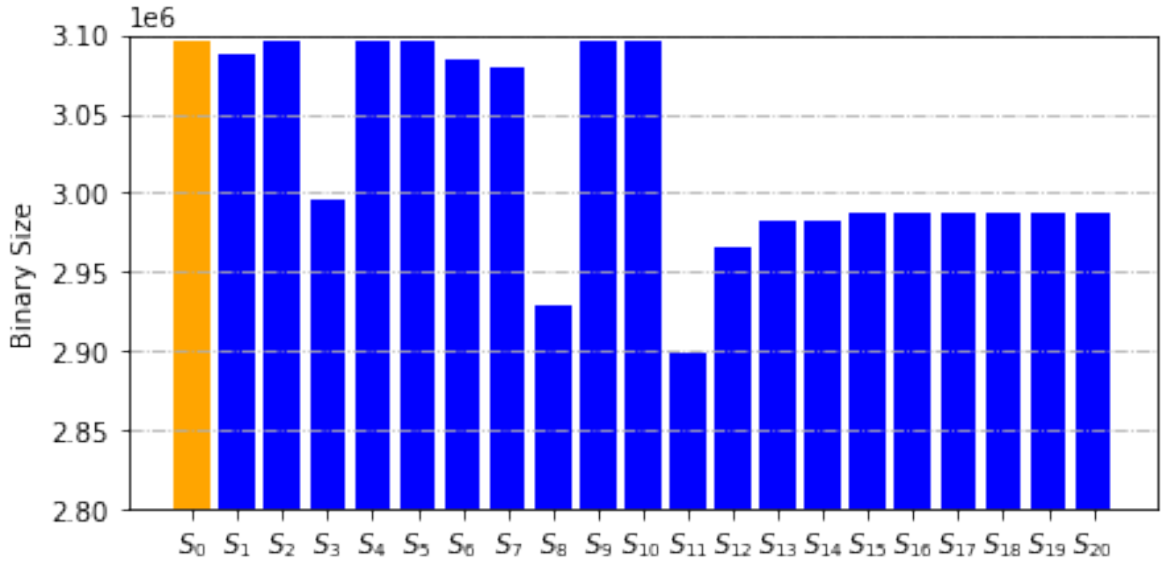
Settings: Workstation: Fedora 33
CPU: Intel Core i7-10610U with 15.3 GiB of memory
Compiler: GCC 10.3.1

Repeated: 5 times, sequentially

Hypothesis testing: Wilcoxon signed-rank test ($\alpha = 0.05$, one-sided)

H_{01}, H_{A1}: The binary size of specialized system

$(0.001\% - 5.416\%)$ $(3.526\% - 6.365\%)$ $\rightarrow$ average 2.45%



$S_0 : 3,096,176$ bytes; $S_{1-20} : (3,020,417 \pm 63,641)$ bytes

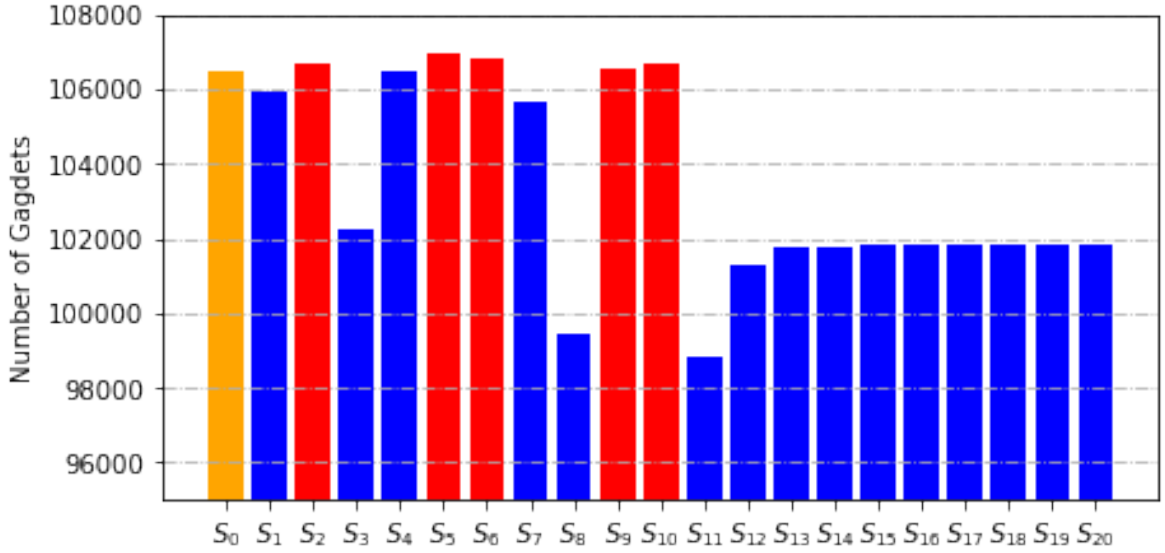
H_{01} is rejected in favor of H_{A1} as $p = 9.54 \cdot 10^{-7} < \alpha = 0.05$.

i.e., specializing a software system regarding its run-time options will statistically significantly reduce its binary size.

H_{02}, H_{A2}: The attack surface of specialized system

Gadgets: small code sequences in a system that end in a RET or JMP return instruction

$(-0.42\% - 6.60\%)$ $(4.53\% - 7.18\%)$ $\rightarrow$ average 2.90%

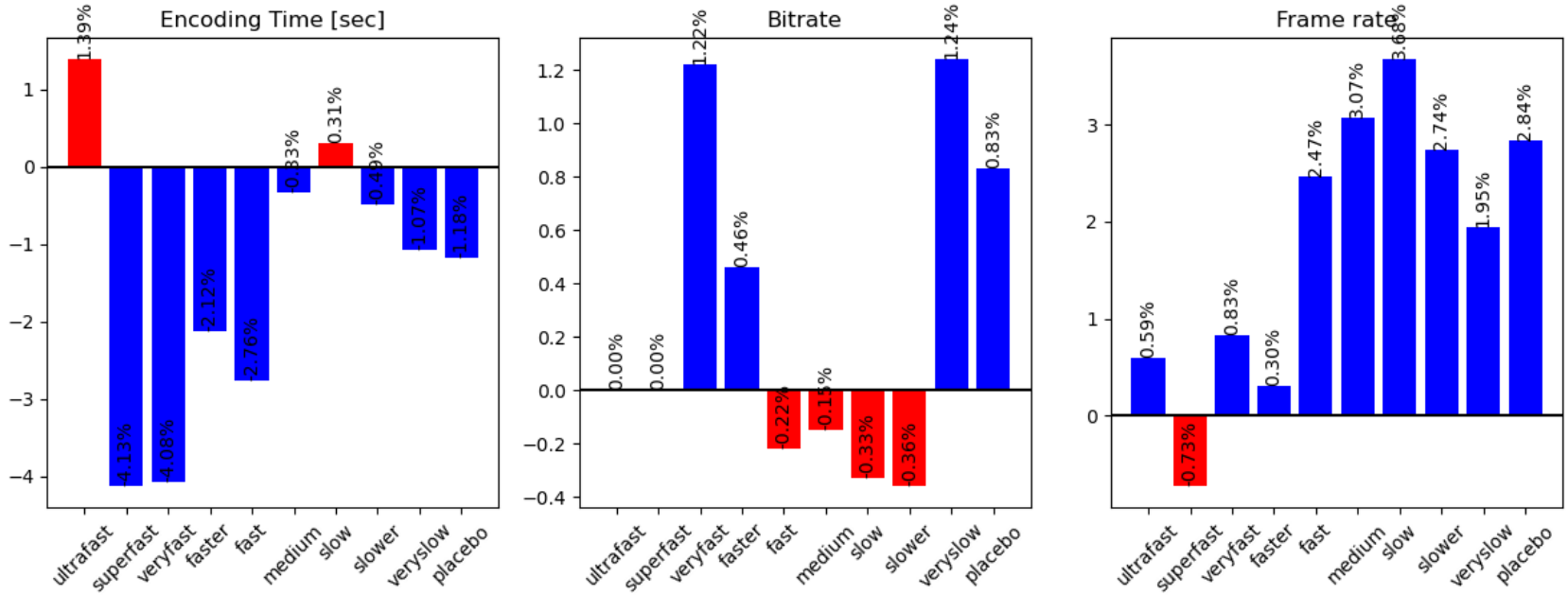


$S_0 : 106,495$ gadgets; $S_{1-20} : (103,414 \pm 2,696)$ gadgets

H_{02} is rejected in favor of H_{A2} as $p = 3.54 \cdot 10^{-4} < \alpha = 0.05$. H_{02} is not rejected for $S_1 - S_{10}$ $(p = 0.25)$ and is rejected for $S_{11} - S_{20}$ $(p = 9.77 \cdot 10^{-4})$.

i.e., it is better to specialize a software system by multiple run-time options, as it will statistically significantly reduce its attack surface.

H_{03}, H_{A3}: The performance of specialized system



H_{03} is rejected in favor of H_{A3} for encoding time ($p = 0.01$) and frame rate ($p = 0.02$), but not for bitrate ($p = 0.16$). H_{03} is not rejected for $S_1 - S_{10}$ ($p = 0.5$, $p = 0.16$, and $p = 0.46$, respectively). But, it is rejected for $S_{11} - S_{20}$ for encoding time ($p = 1.95 \cdot 10^{-3}$) and frame rate ($p = 0.02$).

i.e., specializing a software system regarding its run-time options could significantly improve its performance (in x264, its encoding time and frame rate).

The trade-off among the system properties

System	Binary size	Gadgets	Encoding time	Bitrate	Frame rate
S_1	−0.270%	−0.51%	0.89%	0.00%	−1.78%
S_2	−0.001%	0.18%	−5.38%	0.96%	8.53%
S_3	−3.254%	−4.00%	0.87%	0.00%	−0.65%
S_4	−0.001%	−0.02%	1.06%	0.00%	−1.88%
S_5	−0.001%	0.42%	1.29%	0.00%	−2.26%
S_6	−0.403%	0.29%	1.07%	0.00%	−2.91%
S_7	−0.543%	−0.81%	−4.73%	0.00%	1.86%
S_8	−5.416%	−6.60%	−0.14%	0.00%	2.32%
S_9	−0.003%	0.05%	1.58%	0.00%	2.77%
S_{10}	−0, 001%	0.18%	NaN	NaN	NaN
S_{11}	−6, 365%	−7.18%	1.71%	0.00%	−0.68%
S_{12}	−4.202%	−4.88%	−3.07%	0.00%	−0.91%
S_{13}	−3.659%	−4.43%	−9.58%	2.50%	8.15%
S_{14}	−3.659%	−4.43%	−9.58%	2.50%	8.15%
S_{15}	−3.526%	−4.35%	−11.97%	−0.67%	14.77%
S_{16}	−3.526%	−4.35%	−11.28%	−0.45%	17.64%
S_{17}	−3.526%	−4.35%	−7.24%	−0.98%	18.65%
S_{18}	−3.526%	−4.35%	−5.41%	−1.09%	13.91%
S_{19}	−3.526%	−4.35%	−6.59%	3.71%	12.52%
S_{20}	−3.526%	−4.35%	−4.25%	2.49%	8.41%
Avr.	−2.447%	−2.89%	−3.40%	0.36%	5.47%

Specialization of Run-time Configuration Space at Compile-time: An Exploratory Study

Xhevahire Tërnava, Mathieu Acher, and Benoit Combemale

Contributions:

- A semi-automated and sound system specialization approach
- Quantify benefits: binary size (2.45%), attack surface (2.89%), encoding time (3.40%), bitrate (0.36%), and frame rate (5.47%)
- 4/5 analysed properties show a significant improvement
- Design of the study and experimental protocol
- An available ground truth and data for reproduction and replication: https://github.com/ternava/x264/tree/x264-rmv

Remained challenges:

- Automation of our approach, while maintaining system validation
- Generalisation of the approach and system validation
- Exploring the benefits on other (non-)functional properties