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

Xhevahire Tërnava, Mathieu Acher, Benoit Combemale

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

Xhevahire Tërnava
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
xhevahire.ternava@irisa.fr

Mathieu Acher
Univ Rennes, CNRS, Inria, IRISA
Institut Universitaire de France (IUF)
UMR 6074, F-35000 Rennes, France
mathieu.acher@irisa.fr

Benoit Combemale
Univ Rennes, CNRS, Inria, IRISA
UMR 6074, F-35000 Rennes, France
benoit.combemale@irisa.fr

ABSTRACT

Numerous software systems are highly configurable through run-time options (*e.g.*, command-line parameters). Users can tune some of the options to meet various functional and non-functional requirements such as footprint, security, or execution time. However, some options are never set for a given system instance, and their values remain the same whatever the use cases of the system. Herein, we design a controlled experiment in which the system's run-time configuration space can be specialized at compile-time and combinations of options can be removed on demand. We perform an in-depth study of the well-known `x264` video encoder and quantify the effects of its specialization to its non-functional properties, namely on binary size, attack surface, and performance while ensuring its validity. Our exploratory study suggests that the configurable specialization of a system has statistically significant benefits on most of its analysed non-functional properties, which benefits depend on the number of the debloated options. While our empirical results and insights show the importance of removing code related to unused run-time options to improve software systems, an open challenge is to further automate the specialization process.

KEYWORDS

program specialization, performance, debloating, unused variability

1 INTRODUCTION

Modern software systems are highly configurable and expose to the users their abundant configuration options. By setting configuration options' values, a software system is customized for different users to reach specific functionality or performance goals (*e.g.*, execution time, energy consumption, quality of the result) without the need to modify its source code. Such systems have a large and different number of options. A considerable number of their options are set at run-time, which are typically accessible via command-line parameters, configuration files, or menu preferences. Such run-time variability is a strength since system's users have the flexibility to tune their systems, owing to their specific use case. However, there is evidence that "*a significant percentage (up to* 54.1*%) of parameters are rarely set by any user*" [51], that is, the options' values remain the same for a given usage, for example, depending on the user, the performed activities, and the targeted environment. Thus, they unnecessarily "bloat" the software. For instance, a user or an organization may choose always to encode a video with `x264` in an (ultra-)fast way. In this case, its run-time configuration option of `--cabac` is always disabled, as it offers better compression, but requires more processing power to encode and decode a video. Such unused functionalities in a software system can threaten its security, slow down its performance, affect its reliability, or increase its maintenance [36]. Thus, run-time variability is sometimes unnecessary and does not have to be embedded.

Based on this observation, we bring up the idea of *specializing the run-time configuration space* of a software system. The goal is to retain only a subset of configurations that meet a functional or a performance requirement and thus discard the rest. Specifically, we aim to debloat run-time options that are never used within the source code, at compile-time. Which options are unused depend on the system's usage context and they are inputs to the specialization process. Our hypothesis is that the code of unused run-time options is a manifestation of code bloat that may increase the binary size, the attack surface, or slow down the system. For instance, options that are never used induce dead code that can be eliminated (*e.g.*, by a compiler), thus improving the overall system. This paper aims to investigate this hypothesis and explore to what extent removing run-time options can bring benefits to the non-functional properties of software. To the best of our knowledge, quantifying the benefits (if any) has received little attention.

Specializing the configuration space of a software system has a long tradition since the seminal paper of Czarnecki *et.al.*, [13, 14] and others (*e.g.*, [1, 3, 11, 19, 28, 46]). However, most of the works are focused on the specialization of variability models where constraints among individual options (or across several options) are added to enforce the configuration space. A missing part we investigate in this paper is to propagate this specialization to the source code of a configurable system. Specialization can also be seen as a debloating problem where a subset of run-time options to specialize are bloat of a configurable system. Several debloating approaches are proposed in the literature. Most of them provide a way for debloating the unused functionalities from the external libraries or from high-level "features" of a system [37, 40, 43]. Yet, to the best of our knowledge, debloating run-time options of the configurable systems has received little attention. An important specificity of our problem is that the specialization should be flexible and is itself configurable. That is, not all options are set once and for all: in contrast, existing debloating techniques specialize the system under study with a full configuration. Our proposal is to annotate run-time options with compilation directives throughout the source code. As such, the configuration space of run-time options can be specialized at compile-time and options can be removed on-demand.

To realize the idea of specializing a configuration space, several new challenges need to be addressed, such as *(i)* to locate the run-time options within the source code and *(ii)* to take care of the system's validity after its specialization. Our research methodology

is to statically annotate run-time options to mitigate the risk of synthesizing an unsound and incomplete specialization. In this way, this controlled effort limits the introduction of errors that could bias the benefits (if any) of specialization on non-functional properties. In addition, we establish a ground truth for future automatic program specializers and we report on insights from our experience on the case of x264. The contributions of this paper are as follows:

- We propose a means for configuring the specialization of a software system through debloating its unused run-time configuration options at compile-time;
- We analyze the resulting non-functional properties (*i.e.*, binary size, attack surface, and performance) under two specialization scenarios of x264, showing improvements of the x264 configurable system without sacrificing its interoperability and validity;
- We made available the data for reproducing the experiments, the ground-truth for the x264 case study, and call to replicate our study for further confirming or refuting our results [1].

2 BACKGROUND AND MOTIVATION

In this section, we introduce the basic concepts of configurable software systems and the system used as a case study.

2.1 Background

Users can configure a software system by setting the values of its numerous options. Usually, an option carries a particular functionality, has a type of value (namely, Boolean, integer, or string), and a binding time (namely, compile-time or run-time). A *configuration* is an assignment of values to options at their specific binding time. Default values are also assigned to some options. Hence, users can build their custom software by (de-)activating some options at compile-time or run-time, but not only at these times [9]. The interest of compile-time options is to embed in the resulting executable program what is necessary for a given use case. Typically, preprocessor directives (*e.g.*, #ifdef in C/C++ projects) are used to implement them. As for the run-time options, they are (de-)activated prior to, or even during, the execution. Command-line parameters, plugins, or configuration files are examples of mechanisms that users can rely on to control them. In this work, we considered the run-time options, which are set as command-line parameters during the software's load-time. Within the source code, run-time options are implemented by ordinary control statements, such as *if* statements, and locating them in code can be challenging as they are implicit. Nowadays, software systems such as x264 video encoder are configurable by their compile-time and run-time options.

2.2 Motivating case study

As a motivating system for this study is chosen the software system of x264 [2]. It is a command-line video encoder implemented in C, which has been used for the past 5 years to evaluate the debloating approaches [42], is studied among the highly configurable systems (*e.g.*, [2, 17, 20, 24, 41, 46]), and has plenty of run-time options. We use the x264 at its recent commit db0d417 [3]. It has 114,475 lines of code (LoC) [4], 237 files, 39 compile-time options, and 162 run-time options. For example, it is possible to deactivate the support of mp4 format at compile-time by using the --disable-lsmash option during the system build, as in the following.

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

After the system's build, the run-time options can be set. They have an effect on the properties of the encoded video, namely on its encoding time, bitrate, and frame rate. Here is a possible usage of three run-time options in x264.

```
$ x264 --cabac --mbtree --mixed-refs -o vid.264 vid.y4m
```

The used --cabac, --mbtree, and --mixed-refs options are Boolean. They also have their variant to negate their functionality, for example, --mbtree has --no-mbtree.

x264 is a free software application for encoding video streams into the H.264 compression format. It is often used as a library in large systems, such as in space flight hardware [5] where its *(i)* binary size and *(ii)* performance matters. Besides, *(iii)* a vulnerability in the x264's (H.264 decoder) function could be used to attack a larger system, such as the case with the Cisco Meeting Servers [6]. Nowadays, the security of modern software systems is mostly threatened internally, that is, by reusing their existing code, without the need for code injection [40]. This kind of attack allows an attacker to execute arbitrary code on a victim's machine. In this attack, the attacker is able to chain some small code sequences (called *gadgets*) and threaten the security of the system. Basically, the exploited code sequences by the attacker end in a return (RET or JMP) instruction. Therefore, one of the commonly used metrics for measuring the attack surface in a system is *the number of code reuse gadgets* that are available and which can be exploited by an attacker [10, 23, 40]. Motivated by similar requirements, we analyse the effects of specializing x264 on these three non-functional properties, namely on its binary size, attack surface, and performance.

3 SPECIALIZATION APPROACH

3.1 The vision for debloating

At the system level, the usual approach is to keep as much variability as possible. All run-time options may somehow be needed one day. Furthermore, packages, binaries, or build instructions force users to take them all. However, this variability is sometimes unnecessary: in a given context, some options may never be tuned and thus they always have the same value for all actual use cases of a given instance of the software. For instance, --cabac might always have the *true* value. In this case, its corresponding variable in the source code becomes a constant. By setting the option of --no-cabac will speed up the video encoding time, but the video quality may deteriorate. Hence, the tune of run-time options depends on the objectives and constraints of the user. Keeping an option while actually always using the same value leads to missed opportunities. First, by knowing that an option has the same value,

[1] Companion page: https://github.com/ternava/x264/tree/x264-rmv
[2] http://www.chaneru.com/Roku/HLS/X264_Settings.htm
[3] x264 software system: https://github.com/mirror/x264
[4] Measured using the cloc1 tool: https://github.com/AlDanial/cloc
[5] AVN443 encoder: https://www.tvbeurope.com/production-post/visionary-provides-hd-encoders-for-international-space-station
[6] A vulnerability: https://www.cvedetails.com/cve/CVE-2017-12311/

compilers do not have to make some assumptions and the generated code can be improved. Specifically, compilers can further apply optimizations and constant folding, including the propagation of constants and the removal of dead code. Secondly, the presence of unused options at run-time is likely *(i)* to increase the executable binary size, since the related code will be included; *(ii)* can increase the attack surface; and *(iii)* even slow down the system. The vision is that the proactive removal of the unused run-time options can be beneficial to several non-functional properties of software.

To technically support this vision, developers should have the means to debloat run-time options, for example, by lifting them at compile-time, and thus specialize the original program. With configurable specialization, developers can build specialized packages (binaries) out of the specialized program. For example, one can envision delivering new variants of `x264`: `x264-fast`, with fast encoding time; `x264-hq`, with high quality video encoding; `x264-secured`, with fewer code-reuse gadgets; or `x264-tinyfast`, a small binary size yet fast `x264`. Importantly, they do not necessarily specify a full and complete configuration: not all options are subject to specialization since a part of the variability can be preserved at run-time for addressing several usages. But, some of the unneeded options can be eliminated from code to improve the system's non-functional properties. Inspired by multi-staged configuration of variability models [12, 19], the specialization in our case occurs at the code level and regarding the run-time options.

3.2 Process for program specialization

To quantify the benefits of such a vision, we report in this paper about our experiments in debloating run-time options of `x264` through program specialization. To achieve our vision, one should be able to specialize `x264` by removing run-time configuration options on demand. This implies removing the parts of source code that implement such options, which is different from simply disabling those options in a configurator or at the command-line parameter level. Moreover, to be able to remove the unused run-time options, they first need to be located in the source code. Hence, we propose a removal approach which is based on 4 tasks: (`T1`) *we first comprehend the implementation of run-time options*, then (`T2`) *we annotate the corresponding code which implement a given run-time option*, and finally (`T3`) *we apply a partial evaluator* taking care of the actual code removal. In addition to these three tasks, we conducted the last task (`T4`) which consists in *validating the program specialization* by comparing the resulting debloated program with the original one according to a given run-time configuration.

The experiments conducted for these 4 tasks in the context of run-time options in C-based systems, taking `x264` as an illustrative example, are presented in the following. As a partial evaluator, we decided to use the preprocessor switches [6] and all the optimizations (*e.g.*, dead code elimination) provided by GCC. A *partial evaluator* is a method for simplifying programs when program parameters are known [6, 21]. For instance, when it is known that a run-time configuration option in a given system will always be unset. In this study, the program parameters are the given run-time options for removal whereas their removal and code optimization is completed by the GCC compiler. Hence, the annotations of the source code consist in adding the required C preprocessor directives to let GCC remove the code corresponding to the given run-time options. Moreover, the validation of the program specialization is not realized at the program level but at the binary level after the complete compilation of the original program.

`T1`*: Comprehending the implementation of run-time options.* To debloat unused run-time options later, we first had to comprehend their implementation within the codebase. During this process, we identified the following patterns how they are handled in the source code. (A) The `x264`, and most of the C-based systems, uses the `getopt.h` [7] which has the functionalities to automate some of the chores involved in parsing typical `Unix` command-line options. (B) A corresponding variable is usually used to represent a run-time option. In this case, we manually analyse the data-flow corresponding to this variable and locate all parts of the code used to implement that particular run-time option. (C) Some functions were dedicated to handling a single run-time option. Hence, removing all the functions calls will eventually lead to the removal of the complete functions considered as dead code. (D) Then, some of the options have dependencies, for example, `--cabac` and `--no-cabac` have an alternative logic. This suggests that `x264` will be unsound if both these options are removed.

Adding annotations and the experiment's set up took less than 12 person-weeks to be finished. We recorded patterns (A)-(D) to automate the removal of unused runtime options in the future.

`T2`*: Annotating source code.* Our approach of removing unused run-time options is as in the following. First, (1) we lift run-time options to compile-time through annotating them with preprocessor directives. Then, (2) the objective is to let the preprocessor and optimizations of GCC remove the source code related to the values of a given run-time option. A way to make explicit options' values is to annotate them in places where they are implemented (*i.e.*, to locate them). For this purpose, we propose to use C preprocessor directives, namely, `#define`, `#if`, and `#endif`. They are special instructions directed to the preprocessor, which is part of the compiler, in order to be processed. Because of their prominent and widespread nature, we chose them as they are pretty light to learn, use, and make it possible to remove the enclosed code.

Algorithm 1 shows the static approach that we conducted to delimit Boolean run-time options in `x264`, one at a time. The following example illustrates it. In `x264`, both values of a Boolean run-time option use different names, such as `--cabac` and `--no-cabac`, therefore we will refer to them as run-time options instead of values of an abstract option. In Listing 2 is shown a code snippet that implements these options. It has a conditional `if` statement with two branches where both options `--cabac` and `--no-cabac` are involved. When the condition is `true` (lines 4 – 5), that is, the user chose to encode a video with the option of `--cabac`, then the function on the second branch for `--no-cabac` (line 11) becomes dead code and thus can be removed. Further, a compiler can realize that some functions are never called and will further optimize the code.

As *Algorithm 1* shows, to locate and remove the related code of an unused option we first make sure that the system of `x264` is compilable. We then add the `removeoption.h` file and two directives, namely `CABAC_YES` and `CABAC_NO`, as shown in Listing 1. Further,

[7] https://www.gnu.org/software/libc/manual/html_node/Getopt.html

Algorithm 1 Delimiting a Boolean run-time option

Input: compilable θ C-based system with run-time options
Output: compilable θ with delimited run-time option(s)
Create: $removeoption.h \in source\_files$ in θ

```
for option ∈ θ_runtime_options do
    for option = true, false do                    // e.g., true = --cabac, false = --no-cabac
        Within removeoption.h
            #define OPTION_YES 1                   // e.g., #define CABAC_YES 1
            #define OPTION_NO 1                    // e.g., #define CABAC_NO 1
            Set OPTION_YES 0 ⇒ θ_specialized ⊂ θ
        repeat
            Find code that implement its true value
            Enclose it within #if OPTION_YES...#endif
        until θ_specialized is valid
            Set OPTION_YES 1,
            Set OPTION_NO 0 ⇒ θ_specialized ⊂ θ
        repeat
            Find code that implement its false value
            Enclose it within #if OPTION_NO...#endif
        until θ_specialized is valid
            Set OPTION_NO 1
    end for
end for
```

each of them are used to annotate all lines of code related to options --cabac and --no-cabac, respectively, as is given in Listing 2 in lines 3, 6 – 7, 9 – 10, and 12. We used preprocessor directives at different granularity levels and in different combinations. In Listings 2 and 3 is shown their usage to annotate the related code of --cabac and --no-cabac at statement and function levels, respectively.

In this way, we continue to annotate statically the related parts of code for one option at a time until the system of x264 becomes again compilable and valid. We check the validity of the program when an option is removed, that is, of its specialized version.

It should be emphasized that, while we add the run-time options' annotations, we kept unmodified the existing code of the system. For this reason, we add two directives in combination as in lines 7 and 9 in Listing 2. This way of annotating allows the user to remove one of the options by setting the values $(0, 1)$ or $(1, 0)$ in Listing 1 for each directive respectively. Setting both directives to 0 is unacceptable as at least one of the options must be part of the system. Whereas, when both directives are set to 1 then both options are kept available at run-time.

The algorithm is repeated for as many Boolean options as it is necessary and it can be extended for not Boolean options.

The process ends as soon as the system after the option's removal is valid. The part of the approach to ensure system validity and doing its automated specialization are given in the following.

T3 & T4: *Debloating validation.* After annotating and removing each option, we check the validity of x264, that is, if the the remained system delivers the same functionality as before when the option was part of the code but unset/disabled (lines 10 and 16 in Algorithm 1). The validity check is automated and has three steps.

```
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 */
```

Listing 1: An excerpt from the `removeoption.h` file with 2 defined preprocessor directives

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

Listing 2: The `encoder.c` file where --cabac and --no-cabac are implemented

```
1  /* File common/cabac.c */
2  /*The previous code is omitted*/
3  #if CABAC_YES
4  void x264_cabac_init( x264_t *h ) {
5      int ctx_count = CHROMA444 ? 1024 : 460;
6      for( int i = 0; i < 4; i++ ) {
7      // 8 lines of computation
8      }
9  }
10 #endif
11 /*The rest of the code is omitted*/
```

Listing 3: The `cabac.c` file where the directive is used at the function level

First, the original and specialized program are used to encode eight carefully chosen videos (*cf.* Section 4.4) by the ten built-in presets [8] in x264 and record the resulting video sizes (A). In the next check (B), if the size in bytes of each encoded video by the original and specialized program is exactly the same and if using the removed option (*e.g.*, of --cabac) at run-time in the specialized program is not possible anymore we then conclude that the specialized program is valid [9]. In case that at least one of these two conditions is false, then the program is invalid, meaning that the annotated option (*e.g.*, of --cabac) needs to be further improved. Finally, we check the interoperability of the specialized program (C). For example, trying to encode a video by setting the removed option of --cabac must show a warning message which notifies the user that the option of --cabac is no longer available. The validation process is specific to our program of interest, here x264. However, the methodology is general and essentially requires to develop a testing procedure (*e.g.*, oracle [5]) capable of comparing two programs (*e.g.*, their outputs) – the original and the specialized ones.

4 EXPERIMENTAL DESIGN

4.1 Hypothesis

We present three null hypotheses each to be supported or refuted by our analysis. H_{01} concerns the binary size, H_{02} the attack surface, and H_{03} the performance of a software system. Their respective alternatives are H_{A1}, H_{A2}, and H_{A3}.

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.

[8] A preset is a set with run-time configuration options that are set at once.
[9] Comparing the videos' content is used too: https://github.com/ternava/x264/discussions/15

Table 1: Run-time options and presets to specialize x264

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}	●	●	●	●	●	●	●	●	●	●

Table 2: The properties of the eight used videos

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

H_{A2} : The specialized software system has smaller attack surface (*i.e.*, # code reuse gadgets) 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.

To statistically evaluate our hypotheses, we build 20 specializations of x264. Then, we measure their binary size, attack surface (*i.e.*, # code reuse gadgets), and performance (*i.e.*, encoding time, bitrate, and frame rate) using 8 input videos. Finally, we do a statistical hypothesis testing by using the Wilcoxon signed-rank test [50], which is one of the most used statistical test in software engineering [15]. We reject any of the null hypothesis at a common significance level of $\alpha = 0.05$ and do one sided ("greater"/"less") testing.

4.2 Baseline system

The system of x264 comes with a default configuration (*cf.* its analysed commit, Section 2.2), that is, each of its compile-time and run-time options is either enabled or disabled by default. It uses 6 external libraries as compile-time options, which are enabled by default. We installed all of them. Aside from this, the other compile-time and run-time options are kept to their default values. We refer to such an x264's version with default configuration as the *baseline system*. In the following, we will refer to its measured properties as the *baseline binary size*, *baseline number of gadgets*, *baseline encoding time*, *baseline bitrate*, and *baseline frame rate*.

4.3 System specialization

The sample of run-time options. As given in Section 2.2, x264 has 162 run-time configuration options and 10 built-in presets. A preset is a group of run-time options that are commonly set to quickly encode a video. In x264, only 22 options are used to build its 10 presets. For this exploratory study, we chose a sample of 10 run-time options in x264 that are mainly part of these presets and may help thus to address our hypothesis. In Table 1 are given the 10 available presets in x264 and the presence of the 10 chosen run-time options into them. In particular, we chose 9 options that are part of at least one preset, 1 option that is not a part of any preset (--no-psy), and 1 option that is not part of the presets (--psy). For example, the preset ultrafast can be used as in the following and it includes up to 17 options.

```
$ x264 --preset=ultrafast -o <output_video> <input_video>
```

We have chosen to analyze 5 of its options, which will be set at once by this preset and hence marked with filled circles (●) in Table 1. The 5 other chosen options, namely --mixed-refs, --cabac, --mbtree, --weightb, and --no-psy, we have chosen to specialize x264 regarding this preset. These options are marked with unfilled circles (○) in Table 1, meaning that they will remain unset in the case when users use x264 to always encode videos with the ultrafast preset. In the companion page we provide a table with details regarding the set and unset options' directives in each specialization.

Independent and dependent variables. To conduct our experiment, we have as control variables: the compile-time options in the subject system, the number and kind of used videos, and the used environment. Then, we can identify the independent variable: the remained run-time options in x264 after its specialization. The dependent variables for the study were the measures of binary size, number of code reuse gadgets, encoding time, bitrate, and frame rate.

Specialization scenarios. By using the 10 chosen run-time options, we follow two scenarios for specializing x264.

Scenario$_1$: **x264 is specialized by removing one run-time option at a time.** With the 10 considered options, there are only 10 such possible specializations of x264. They consist of the specializations regarding one of the 10 run-time options (*i.e.*, horizontal aspect) given in the first column of Table 1.

Scenario$_2$: **x264 is specialized by removing one preset at a time.** In this case, we specialize x264 regarding the 10 presets by removing the set of unused options within each specific preset. Based on Table 1, they consist of the vertical specializations. For instance, towards an x264-fast version, x264 will be specialized regarding the fast preset by removing its 5 unused options, which are marked with "○". Hence, in this scenario, there are only 10 other possible specializations of x264. But, they can be grouped into four specializations, for example, veryfast and faster have the same used and unused options and thus both specializations look the same. Despite this, we still analyze them separately as their performance may differ because of their other (un)set options that are different and are kept out of the study. For instance, ultrafast has 17 options, but only 10 of them are analyzed.

In total, there is the baseline of x264 and its 20 possible specializations with both scenarios. In the following, we will refer to them as S_0, for the baseline, and S_1 –S_{20}, for the specialized systems.

4.4 Experiment settings

We conducted our experiment on a Linux workstation running Fedora 33 with Intel Core i7-10610U CPU and 15.3 GiB of memory. In all cases, we use the gcc 10.3.1 compiler to build the baseline and specialized systems of x264. Some of the done measurements (*cf.* H_{03}) require an input, namely a video. Instead of a random

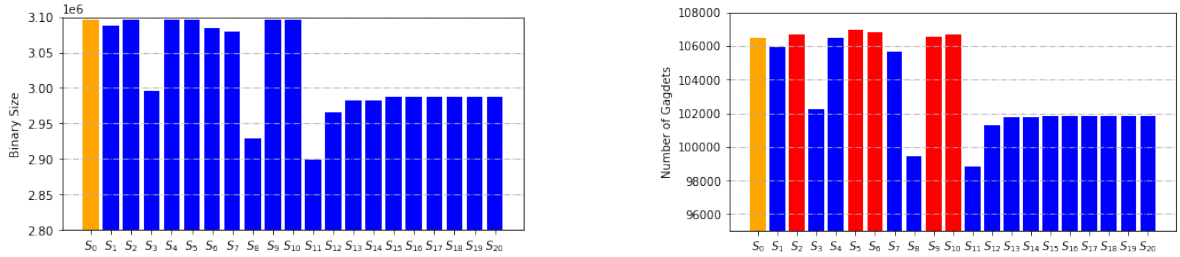

Figure 1: The binary size and the found number of gadgets in S_0 (baseline) and $S_1 - S_{20}$ (specializations). $(mean \pm std)$ **for binary size -** $S_0 : (3,096,176 \pm 0)$**,** $S_{1-20} : (3,020,417 \pm 63,641)$**, and for gadgets -** $S_0 : (106,495 \pm 0)$**,** $S_{1-20} : (103,414 \pm 2,696)$

selection, we carefully chose 8 videos based on a recent work by Lesoil *et.al.*, [25], which properties are given in Table 2. Actually, these video are evaluated to be as representative enough of 1,300 input videos in the YT UCG dataset [10] [25]. Besides, considering the suggestions by *DB. Stewart* [44], the encoding time of videos is measured by using the *time* method, and the measurements are repeated 5 times. To prevent side effects, all experiments are run sequentially and as the only process in the workstation.

5 RESULTS

In this section, we present the obtained results on three hypothesis.

5.1 The binary size of specialized system

To address H_{01}, we measured the binary size of `x264` in bytes. At first, under the same experiment settings, we measured the baseline binary size of `x264`. Then, we specialized `x264` following $Scenario_1$ and $Scenario_2$, measured their respective binary size, and compared them with the baseline binary size. Finally, we statistically test the H_{01} null hypothesis.

The obtained results in Figure 1 (left) show that each specialized system has a smaller binary size than the baseline system. The smallest binary size have the specialized systems by $Scenario_2$ (S_{11} - S_{20}), as they are specialized by more than one run-time option, than those by $Scenario_1$ (S_1 - S_{10}). In percentage, compared to the baseline binary size, the specialized systems S_1 - S_{10} have a reduced binary size between 0.001% and 5.416%, whereas S_{11} - S_{20} have a reduced binary size between 3.526% and 6.365%. Concretely, the `x264`'s binary size is reduced by 2.45% on average, or up to 6.37% (based on 10 analysed options). To avoid repetition here, all of these data in percentage are also given later in Table 5.

Moreover, the calculated Wilcoxon signed-rank test shows that the p-value is less than $\alpha = 0.05$ ($t = 210$, $p = 9.54 \cdot 10^{-7}$) [11], therefore H_{01} is rejected in favor of H_{A1}. This suggests that specializing a software system regarding its run-time options will statistically *significantly* reduce its binary size. This reduction can be useful for resource-constraint devices. Although there is no direct or strong correlation between binary size and other non-functional properties, the reduction suggests that many paths of the code are eliminated and can be beneficial to the system to be run.

5.2 The attack surface of specialized system

To measure the attack surface, we counted the number of code reuse gadgets in the baseline and 20 specialized systems of `x264`. For this reason, we used the well-known ROPgadget [12] tool and counted the overall gadgets in the system's binary. Then, we statistically test the H_{02} null hypothesis.

The resulting overall number of code reuse gadgets in Figure 1 (right) show that the overall number of gadgets are most often reduced (the blue bars) compared to the baseline number of gadgets (the orange bar). The specialized systems S_1 - S_{10} have fewer gadgets than the baseline system, between −0.42% and 6.60%, whereas specialized systems S_{11} - S_{20} have fewer gadgets, between 4.35% and 7.18%. The −0.42% indicates that some specialized systems by a single option (specifically, the 5 red ones, see also Table 5) can have a small increase in the number of gadgets despite that their binary size is reduced. This is something that is also claimed by [10] that could happen, that is, the debloating techniques can cause an increase in the number of gadgets as a side effect. But, not all gadgets in a system can be chained by an attacker and threaten the system.

However, the calculated Wilcoxon signed-rank test shows that the H_{02} is not rejected for S_1 - S_{10} ($t = 35$, $p = 0.25$), whereas it is rejected for S_{11} - S_{20} ($t = 55$, $p = 9.77 \cdot 10^{-4}$). In general, it is rejected for S_1 - S_{20} in favor of the H_{A2} ($t = 190$, $p = 3.54 \cdot 10^{-4}$). This suggests that it is better to specialize a software system by multiple run-time options (*i.e.*, $Scenario_2$), as this will statistically *significantly* reduce its attack surface (by 2.9% on average, or up to 7.18% based on 10 options in `x264`), and hence its security will be improved. Moreover, the attack surface gets reduced much quicker, for 2.9% on average, than the binary size, for 2.5% on average.

5.3 The performance of specialized system

Next, we examine how a specialized system performs to the end-users. Namely, as `x264` is a video encoder, we measure the performance of a specialized `x264` by using three metrics: the video *encoding time*, *bitrate*, and *frame rate*. For this purpose, we encoded 8 videos (*cf.* Table 2) using the baseline and each of the 20 specializations of `x264`, but only using their available presets. For example, S_6 is specialized regarding the `--mixed-refs` (*cf.* Table 1). We should encode videos only with 4 presets in it, namely, with `ultrafast`, `superfast`, `veryfast`, and `faster`. The rest of the presets should not be used in S_6 as they require the option of `--mixed-refs`, which

[10] UCG dataset: https://media.withyoutube.com/

[11] Where t is the sum of the ranks of the differences above or below zero.

[12] ROPgadget tool: https://github.com/JonathanSalwan/ROPgadget

Table 3: The average and standard deviation of *encoding time* in seconds [s] of 8 videos by the baseline (S_0) and specialized systems ($S_1 - S_{20}$). The R and $\neg R$ mark a rejection and nor rejection of the H_{03} hypothesis, respectively.

System		ultrafast	superfast	veryfast	faster	fast	medium	slow	slower	veryslow	placebo	H_{03}
S_0	[s]	0.28 ± 0.02	0.48 ± 0.01	0.74 ± 0.03	1.03 ± 0.04	1.41 ± 0.04	1.73 ± 0.04	2.76 ± 0.08	4.97 ± 0.16	9.85 ± 0.37	35.03 ± 0.72	
S_1	[s]	-	-	-	-	1.39 ± 0.03	1.77 ± 0.05	2.85 ± 0.10	5.08 ± 0.18	9.86 ± 0.26	35.32 ± 0.46	
S_2	[s]	-	-	0.67 ± 0.02	0.96 ± 0.01	1.29 ± 0.02	1.63 ± 0.04	2.61 ± 0.06	4.78 ± 0.18	9.27 ± 0.42	33.23 ± 0.69	
S_3	[s]	-	0.46 ± 0.02	0.73 ± 0.02	1.02 ± 0.02	1.43 ± 0.03	1.79 ± 0.05	2.80 ± 0.06	5.00 ± 0.18	9.96 ± 0.26	35.34 ± 0.76	
S_4	[s]	-	0.46 ± 0.01	0.73 ± 0.01	1.03 ± 0.01	1.43 ± 0.03	1.78 ± 0.05	2.92 ± 0.10	5.07 ± 0.19	10.11 ± 0.32	35.09 ± 0.48	
S_5	[s]	0.29 ± 0.01	0.46 ± 0.02	0.75 ± 0.01	1.06 ± 0.03	1.46 ± 0.04	1.83 ± 0.04	2.87 ± 0.08	5.05 ± 0.19	10.09 ± 0.37	35.20 ± 0.63	
S_6	[s]	0.30 ± 0.01	0.47 ± 0.02	0.75 ± 0.02	1.05 ± 0.01	-	-	-	-	-	-	
S_7	[s]	0.28 ± 0.01	0.45 ± 0.01	-	-	-	-	-	-	-	-	
S_8	[s]	0.28 ± 0.01	-	-	-	-	-	-	-	-	-	
S_9	[s]	0.29 ± 0.02	-	-	-	-	-	-	-	-	-	
S_{10}	[s]	-	-	-	-	-	-	-	-	-	-	
Avr. S_{1-10}	[s]	0.29 ± 0.01 ↑	0.46 ± 0.01 ↓	0.72 ± 0.03 ↓	1.02 ± 0.04 ↓	1.40 ± 0.07 ↓	1.76 ± 0.08 ↑	2.81 ± 0.12 ↑	4.99 ± 0.13 ↑	9.86 ± 0.34 ↑	34.84 ± 0.90 ↓	¬R
S_{11}	[s]	0.29 ± 0.01	-	-	-	-	-	-	-	-	-	
S_{12}	[s]	-	0.47 ± 0.02	-	-	-	-	-	-	-	-	
$S_{13,14}$	[s]	-	-	0.66 ± 0.03	0.95 ± 0.03	-	-	-	-	-	-	
S_{15-20}	[s]	-	-	-	-	1.24 ± 0.04	1.53 ± 0.03	2.56 ± 0.07	4.70 ± 0.16	9.21 ± 0.40	33.55 ± 0.54	
Avr. S_{11-20}	[s]	0.29 ± 0.0 ↑	0.47 ± 0.0 ↓	0.66 ± 0.0 ↓	0.95 ± 0.0 ↓	1.24 ± 0.0 ↓	1.53 ± 0.0 ↓	2.56 ± 0.0 ↓	4.70 ± 0.0 ↓	9.21 ± 0.0 ↓	33.55 ± 0.0 ↓	R
Avr. S_{1-20}	[s]	0.29 ± 0.01 ↑	0.46 ± 0.01 ↓	0.71 ± 0.04 ↓	1.01 ± 0.05 ↓	1.37 ± 0.09 ↓	1.72 ± 0.12 ↓	2.77 ± 0.15 ↑	4.95 ± 0.16 ↓	9.75 ± 0.41 ↓	34.62 ± 0.96 ↓	R
% of S_{1-20}		1.39%	−4.13%	−4.08%	−2.12%	−2.76%	−0.33%	0.31%	−0.49%	−1.07%	−1.18%	

Table 4: The average and standard deviation of *bitrate* in [kb/s] and *frame rate* in [fps] of 8 videos by S_0 and $S_1 - S_{20}$

System	Unit	ultrafast	superfast	veryfast	faster	fast	medium	slow	slower	veryslow	placebo	H_{03}
S_0	[kb/s]	1, 800 ± 0	1, 167 ± 0	905 ± 0	946 ± 0	947 ± 0	935 ± 0	928 ± 0	851 ± 0	843 ± 0	787 ± 0	
Avr. S_{1-10}	[kb/s]	1, 800 ± 0 ↔	1, 167 ± 0 ↔	912 ± 15 ↑	948 ± 6 ↑	946 ± 3 ↓	935 ± 2 ↓	926 ± 4 ↓	849 ± 4 ↓	849 ± 14 ↑	791 ± 9 ↑	¬R
Avr. S_{11-20}	[kb/s]	1, 800 ± 0 ↔	1, 167 ± 0 ↔	938 ± 0 ↑	959 ± 0 ↑	941 ± 0 ↓	931 ± 0 ↓	919 ± 0 ↓	841 ± 0 ↓	874 ± 0 ↑	806 ± 0 ↑	¬R
Avr. S_{1-20}	[kb/s]	1, 800 ± 0 ↔	1, 167 ± 0 ↔	916 ± 17 ↑	950 ± 7 ↑	945 ± 3 ↓	934 ± 2 ↓	925 ± 5 ↓	848 ± 5 ↓	853 ± 16 ↑	793 ± 10 ↑	¬R
% of S_{1-20}		0.00%	0.00%	1.22%	0.46%	−0.22%	−0.15%	−0.33%	−0.36%	1.24%	0.83%	
S_0	[fps]	2, 257 ± 157	1, 313 ± 43	881 ± 26	590 ± 13	436 ± 9	336 ± 8	218 ± 6	136 ± 5	69 ± 2	18 ± 1	
Avr. S_{1-10}	[fps]	2, 276 ± 53 ↑	1, 303 ± 25 ↓	875 ± 28 ↓	583 ± 23 ↓	436 ± 32 ↑	337 ± 24 ↑	220 ± 18 ↑	136 ± 8 ↑	69 ± 4 ↓	18 ± 1 ↑	¬R
Avr. S_{11-20}	[fps]	2, 242 ± 0 ↓	1, 301 ± 0 ↓	957 ± 0 ↑	634 ± 0 ↑	500 ± 0 ↑	396 ± 0 ↑	259 ± 0 ↑	154 ± 0 ↑	77 ± 0 ↑	19 ± 0 ↑	R
Avr. S_{1-20}	[fps]	2, 271 ± 49 ↑	1, 303 ± 23 ↓	888 ± 42 ↑	591 ± 29 ↑	446 ± 39 ↑	347 ± 32 ↑	226 ± 23 ↑	139 ± 10 ↑	70 ± 5 ↑	18 ± 1 ↑	R
% of S_{1-20}		0.59%	−0.73%	0.83%	0.30%	2.47%	3.07%	3.68%	2.74%	1.95%	2.84%	

is removed from them. In case that they are used, the resulting system's performance is expected to be distorted, and thus unrealistic. Therefore, only the first 4 presets (*i.e.*, with "∘") are available during the encoding with S_6. This way of encoding is followed by all 20 specializations. As for the baseline system, we use all 10 presets.

As for the measured values, we first computed the average of values in 5 repeated measurements, that is, the average encoding time, bitrate, and frame rate of each video by each available preset in each system. Next, we computed the average encoding time, bitrate, and frame rate of all 8 videos within each preset and system. Results in Tables 3 and 4 show the average values with the standard deviation of specialized systems by the $Scenario_1$, $Scenario_2$, and their overall average, per preset. The dash ("-") in Table 3 is used to mark the unmeasured values for the unavailable presets in each specialized system. It can be noticed that S_{10} has no values, meaning that it has no available presets as all of them require the option of `--psy`, which is removed for S_{10}. To save space, the results in Table 4 are more condensed, showing only these three averages for bitrate and frame rate. In percentage, they are more detailed in Table 5.

Encoding time. The overall *Average* S_{1-20} in Table 3 shows that in 8 from 10 presets the encoding time is improved or decreased between 0.33% and 4.13%, whereas it is increased or worsened in only 2 presets and for less, for 0.31% and 1.39%. Actually, the increased time in `ultrafast` and `slow` is very small, for 3.9 and 8.4 milliseconds, respectively. Still, as our used system to measure them has a higher resolution (of 1 microsecond), we believe that the increased encoding times are a consequence of our x264 specializations. Looking at the individual specializations, the S_{15} has the most improved encoding time, for 11.97%. This means that because of its removed unused options the preset `fast` in S_{15} becomes even faster. Moreover, the encoding time measurements have always less than 0.96 seconds in terms of standard deviations.

Bitrate. Table 4 shows that the number of presets with an equal, increased, or decreased bitrate is the same. S_{19} has the most increased bitrate, for 3.71%, indicating that the video compression will be most significantly improved by this specialization because of its removed unused run-time options. In our measurements, the bitrate standard deviations are minor, less than 17 [kb/s].

Frame rate. A more notable improvement can be observed in the frame rate in the *Average* S_{1-20}, in Table 4. The frame rate is improved in 9 from the 10 presets, between 0.30% and 3.68%, whereas only in `superfast` it is worsened, for only 0.73%. As for the specializations, the S_{17} has the most increased frames per second, for about 18.65%, meaning that the video compression will be improved

Table 5: A comparison of changes, in %, of five 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%

by this specialization. The frame rate deviates more in our measurements, up to 157 [fps], because by default it is auto-detected.

The key observations on the x264's performance are that removing even only 5 of its unused run-time options will improve the video encoding time in 80% of the presets (for up to 4.13%), will improve the bitrate in 40% of the presets (for up to 1.24%), and will improve the frame rate in 90% of the presets (for up to 3.86%). In practical terms, using a specialized system in those predefined configurations, a video is encoded faster and exhibits a better frame rate and bitrate than the original x264. Moreover, the calculated Wilcoxon signed-rank test given in the last column in Tables 3 and 4 shows that H_{03} is not rejected for S_1 - S_{10} for encoding time ($t = 28$, $p = 0.5$), bitrate ($t = 10$, $p = 0.16$), and frame rate ($t = 26$, $p = 0.46$). But, it is rejected for S_{11} - S_{20} for encoding time ($t = 54$, $p = 1.95 \cdot 10^{-3}$) and frame rate ($t = 7$, $p = 0.02$). In general, it is rejected for all specializations S_1 - S_{20} in favor of H_{A3} for encoding time ($t = 51$, $p = 0.01$) and frame rate ($t = 7$, $p = 0.02$), but not for bitrate ($t = 10$, $p = 0.16$). Hence, specializing a software regarding its unused run-time options could improve its performance (in x264, it does *not significantly* improve its bitrate, but it statistically *significantly* improves its encoding time and frame rate).

5.4 The trade-off among the system properties

The obtained results in Sections 5.1, 5.2, and 5.3 show that different specialized systems of x264 have notably different improvements on two non-functional properties – namely, on the binary size and attack surface – and on two performance measurements – namely, on the video encoding time and frame rate. Therefore, to understand which removed options most significantly improve the system in all or in most of these five aspects (including the cases with an improvement in the bitrate), we compared the changes of these five properties (from the baseline) for each specialized system. Specifically, we compared the changes regarding the binary size and number of gadgets given in Table 1, the encoding time given in Table 3, and bitrate with frame rate given in Table 4. The results in percentage are given in Table 5.

It can be observed that the specialized systems $S_{11} - S_{20}$ have far more improved values in each of the five aspects than the $S_1 - S_{10}$. In fact, the maximum improvements in five aspects are achieved with the specializations $S_{11} - S_{20}$, which percentages are colored in green. The red-colored percentages show that the number of gadgets and frame rate are worsened the most in the set of systems $S_1 - S_{10}$, whereas the encoding time and bitrate are worsened the most in the set $S_{11} - S_{20}$. The percentages in orange show the systems with the smallest improvement on the binary size, as there are only improvements. Whereas the NaN values are the unmeasured values, as all presets use the run-time option --psy which is used to specialize S_{10}, hence measuring them is unrealistic. It is interesting to note that specializations that provide an improvement or not are different. Therefore, one should find a trade-off among them and choose the specialization that best meets their requirements. For instance, in resource-constraint devices and in those where security is more important, but the encoding time and frame rate are flexible, then S_{11} is a good choice. But, in case that encoding time matters more than the other properties then S_{15} best fulfills this criterion. In fact, there are 4 specializations that have an improvement in all five properties - namely, S_{13}, S_{14}, S_{19}, and S_{20}. These specializations can be chosen when an improvement is expected on all five properties. Besides these changes within every single specialized system, the overall average in the last row in Table 5 reveals that each of the five properties gets improved by specializing the x264 system.

The more unused run-time options are removed the greater the benefits gained. Still, these benefits may vary in the system's binary size, attack surface, and performance, therefore finding their trade-offs to a given usage context is necessary.

6 RELATED WORK

We discern the four following areas of related work that are relevant.

Configurable systems and their non-functional properties. Software product line engineering and variability management is a well-established research area that led in the past decade to several techniques for supporting *e.g.*, orthogonal variability management and derivation of custom variants [4, 7, 8, 30, 34]. In practice, variability is often moved from compile-time to load-time (*a.k.a.*, encoding variability) [48]. The consequence is to deploy the whole product line (or configurable system) in the delivered software systems. In our work, we follow the opposite path: we aim to move from run-time to compile-time. We specifically explore the benefits of applying the derivation process at compile-time (instead of at run-time) in order to specialize legacy software systems, with a particular focus on non-functional properties. Besides, there are numerous works about the non-functional properties and performance of configurable systems (*e.g.*, see [17, 20, 26, 33, 41, 47]). The goal of this line of research is to study the performance modeling of configurable systems without removing run-time options and without altering the original source code. In contrast, we provide an approach for debloating a configurable system regarding its run-time options. We also show the impact of their removal in the system's non-functional properties.

Software debloating. Furthermore, software debloating has been previously explored, for example, to reduce the size of deployed containers [39], or the attack surface of specific programs [18, 22,

38, 40, 42]. Often, proposed approaches debloat compiled binaries, or debloat a system by removing its unused libraries or its stale feature toggles. To our best knowledge, none of them debloat software regarding the unused run-time configuration options. Existing approaches are all inheriting of existing works in program specialization [29]. In [22], an approach is proposed to remove feature-specific shared libraries that are needed only when certain configuration directives are specified by the user in configuration files. In contrast, we target run-time options within the source code.

Program slicing. It is another longstanding method for automatically decomposing programs by analyzing their data flow and control flow [49]. Program slicing techniques require the identification of a variable of interest, and the benefits have been demonstrated in the context of white-box activities, such as debugging, testing, maintenance, and understanding of programs. However, as in our study, configuration options are usually given from an external (black-box) view of the program, and it is challenging to relate internal variables to such high-level configuration options. This has been demonstrated in [27]. In practice, we also observed much more complex data and control flows that would limit the applicability of such an approach. Hence we believe that such tools can be helpful to partly *support* the task of annotating run-time options in the source code.

Performance. Performance has been investigated at run-time, *e.g.*, speculative execution [16] and ahead-of-time compilation [35]. Such speedups come in complement to the proposed specialization approach applied at compile-time. Since the specialization is applied at the source level, existing approaches for optimizing compilers come also in complement to a previous debloating. Performance has been also investigated in the context of approximate computing [32], which provides unsound transformations, still useful in the context of possible trade-offs with accuracy. For example, loop perforation [31] is a useful interpolation technique while unrolling relaxable loops (*e.g.*, signal processing). In contrast, we explore a sound, compile-time derivation of a specialized configuration space.

In general terms and in contrast to the related works, we propose in this paper a unique experimental study that combines the system's *variability management* and *configuration debloating* with a specific focus on its *non-functional properties*, including its *performance*. The program slicing techniques with the identified patterns to implement run-time options (*cf.* **T1**) and program transformations are helpful to automate the specialization process in the future.

7 THREATS TO VALIDITY

External threats. With respect to generalization, we use only a single software subject in a specific domain (video compression) that is implemented in C language. `x264` has been studied in several papers on configurable software (*e.g.*, [17, 20, 33, 41, 46]), but not in the context of specializing configurable systems and debloating run-time options. Our exploratory research aims to investigate a problem that has not been studied or thoroughly investigated in the past: the potential benefits of removing run-time options. As with any exploratory case study, we cannot draw any statistically generalizing conclusions from such studies [45]. However, such generalization of findings is not the goal of such studies – instead, we aim to develop an understanding and propose hypotheses about other similar configurable systems.

Internal threats. The performance, such as the video's encoding time by `x264`, may differ depending on the inputs processed. To mitigate this threat, we use 8 videos considered as representatives of 1,300 videos coming from the YouTube UGC dataset. Then, the measurements are repeated 5 times. A threat to validity is also the set of run-time options we consider. Choosing different options may have a different effect on the performance measurements. We have selected a diverse set of options with potentially different impacts on non-functional properties and which are part of the presets that are widely used and represent a variety of usage in `x264`.

Another threat to our experiments is the annotation of run-time options, which can be incomplete and not valid. An incomplete annotated option may impact different properties (*e.g.*, binary size) and break existing functionalities, not related to the removed options. To mitigate this threat, we manually and thoroughly annotated the source code. We systematically reported on progress through GitHub issues and concrete meetings for clarifying some subtle cases about, for instance, the meaning of debloating. In terms of validity, we used oracles that control that some remaining configurations produce the exact same videos. In terms of completeness, we chose to not consider the assembly code of `x264` when annotating options. A few lines of code and options are involved, suggesting little to negligible impact on our results.

8 CONCLUSION AND FUTURE WORK

We proposed an approach for specializing a software system by changing the binding time of its unused run-time options at compile-time. Using the well-known video encoder of `x264`, we experimented with 10 of its options over 8 inputs (videos) and over its 10 built-in presets. The results show that specializing a system regarding its unused run-time options brings statistically significant benefits *w.r.t.* three non-functional properties, namely binary size, attack surface, and performance (except for bitrate). For instance, removing the single option of `--cabac` in `x264` reduces the binary size by about 7%, its attack surface by 8%, and the video encoding time by 3%. The combination of options during the specialization process can lead to further benefits (up to 18% for frame rate). We believe that developers, operationals, and users can use our approach to specify and then remove run-time options that are never used or changed for their specific use cases.

Our exploratory study naturally calls to replicate the approach and results in other software engineering contexts. We also brought a new problem to the community: instead of adding options, the challenge of specialization is to remove code related to options. Another future work direction is to automate our approach and also to measure the change of the system's non-functional properties when some of its compile- and run-time options are both removed.

REFERENCES

[1] Mathieu Acher, Paul Temple, Jean-Marc Jezequel, José A Galindo, Jabier Martinez, and Tewfik Ziadi. 2018. VaryLaTeX: Learning paper variants that meet constraints. In *12th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 83–88.

[2] Juliana Alves Pereira, Mathieu Acher, Hugo Martin, and Jean-Marc Jézéquel. 2020. Sampling effect on performance prediction of configurable systems: A case study. In *International Conference on Performance Engineering*. ACM, 277–288.

[3] Benoit Amand, Maxime Cordy, Patrick Heymans, Mathieu Acher, Paul Temple, and Jean-Marc Jézéquel. 2019. Towards learning-aided configuration in 3D printing: Feasibility study and application to defect prediction. In *13th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 1–9.

[4] Sven Apel, Alexander Von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for product-line verification: Case studies and experiments. In *35th International Conference on Software Engineering*. IEEE, 482–491.

[5] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *Transactions on Software Engineering* 41, 5 (2014), 507–525.

[6] Ira D Baxter and Michael Mehlich. 2001. Preprocessor conditional removal by simple partial evaluation. In *8th Working Conference on Reverse Engineering*. IEEE, 281–290.

[7] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a feature? A qualitative study of features in industrial software product lines. In *19th International Conference on Software Product Line*. ACM, 16–25.

[8] Paulo Borba, Leopoldo Teixeira, and Rohit Gheyi. 2012. A theory of software product line refinement. *Theoretical Computer Science* 455 (2012), 2–30.

[9] Jan Bosch and Rafael Capilla. 2012. Dynamic variability in software-intensive embedded system families. *Computer* 45, 10 (2012), 28–35.

[10] Michael D Brown and Santosh Pande. 2019. Is less really more? Towards better metrics for measuring security improvements realized through software debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test*.

[11] Rafael Capilla and Jan Bosch. 2011. The promise and challenge of runtime variability. *Computer* 44, 12 (2011), 93–95.

[12] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2004. Staged configuration using Feature Models. In *Software Product Lines*. Springer, 266–283.

[13] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Formalizing cardinality-based Feature Models and their specialization. *Software Process: Improvement and Practice* 10, 1 (2005), 7–29.

[14] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. 2005. Staged configuration through specialization and multilevel configuration of Feature Models. *Software Process: Improvement and Practice* 10, 2 (2005), 143–169.

[15] Tore Dybå, Vigdis By Kampenes, and Dag IK Sjøberg. 2006. A systematic review of statistical power in software engineering experiments. *Information and Software Technology* 48, 8 (2006), 745–755.

[16] Freddy Gabbay and Freddy Gabbay. 1996. *Speculative execution based on value prediction*. Technical Report. EE Department TR 1080, Technion - Israel IT.

[17] Jianmei Guo, Krzysztof Czarnecki, Sven Apel, Norbert Siegmund, and Andrzej Wąsowski. 2013. Variability-aware performance prediction: A statistical learning approach. In *28th Int. Conf. on Automated Software Engineering*. IEEE, 301–311.

[18] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *2018 Conference on Computer and Communications Security*. ACM, 380–394.

[19] Arnaud Hubaux, Patrick Heymans, Pierre-Yves Schobbens, Dirk Deridder, and Ebrahim Khalil Abbasi. 2013. Supporting multiple perspectives in feature-based configuration. *Software & Systems Modeling* 12, 3 (2013), 641–663.

[20] Pooyan Jamshidi, Norbert Siegmund, Miguel Velez, Christian Kästner, Akshay Patel, and Yuvraj Agarwal. 2017. Transfer learning for performance modeling of configurable systems: An exploratory analysis. In *32nd International Conference on Automated Software Engineering*. IEEE, 497–508.

[21] Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation*. Peter Sestoft.

[22] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-driven software debloating. In *12th European Workshop on Systems Security*. ACM, 1–6.

[23] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. 2011. Attack surface reduction for commodity OS kernels: Trimmed garden plants may attract less bugs. In *4rth European Workshop on System Security*. ACM, 1–6.

[24] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. Deep software variability: Towards handling cross-layer configuration. In *15th Int. Working Conf. on Variability Modelling of Software-Intensive Systems*. ACM, 1–8.

[25] Luc Lesoil, Mathieu Acher, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. The Interaction between inputs and configurations fed to software systems: An empirical study. *Preprint arXiv:2112.07279* (2021).

[26] Luc Lesoil, Mathieu Acher, Xhevahire Tërnava, Arnaud Blouin, and Jean-Marc Jézéquel. 2021. The interplay of compile-time and run-time options for performance prediction. In *25th International Systems and Software Product Line Conference-Volume A*. ACM, 100–111.

[27] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking load-time configuration options. In *29th International Conference on Automated Software Engineering*. ACM, 445–456.

[28] Hugo Martin, Mathieu Acher, Juliana Alves Pereira, and Jean-Marc Jézéquel. 2021. A comparison of performance specialization learning for configurable systems. In *25th Int. Systems and Software Product Line Conference*. ACM, 46–57.

[29] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. 2001. Specialization tools and techniques for systematic optimization of system software. *Transactions on Computer Systems* (2001), 217–251.

[30] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring differences and commonalities between feature flags and configuration options. In *42nd International Conference on Software Engineering: Software Engineering in Practice*. ACM, 233–242.

[31] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. 2010. Quality of service profiling. In *32nd International Conference on Software Engineering*. ACM, 25–34.

[32] Sparsh Mittal. 2016. A survey of techniques for approximate computing. *Comput. Surveys* (2016), 1–33.

[33] Juliana Alves Pereira, Hugo Martin, Mathieu Acher, Jean-Marc Jézéquel, Goetz Botterweck, and Anthony Ventresque. 2019. Learning software configuration spaces: A systematic literature review.

[34] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag.

[35] Todd A Proebsting, Gregg M Townsend, Patrick G Bridges, John H Hartman, Tim Newsham, and Scott A Watterson. 1997. Toba: Java for applications-A way ahead of time (WAT) compiler. In *COOTS*. "", 41–54.

[36] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the chromium browser with feature subsetting. In *Conference on Computer and Communications Security*. 461–476.

[37] Anh Quach and Aravind Prakash. 2019. Bloat factors and binary specialization. In *3rd Wrk. on Forming an Ecosystem Around Software Transformation*. ACM, 31–38.

[38] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th USENIX Security Symposium*. USENIX Association, 869–886.

[39] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically debloating containers. In *11th Joint Meeting on Foundations of Software Engineering*. ACM, 476–486.

[40] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application specialization for code debloating. In *33rd International Conference on Automated Software Engineering*. ACM, 329–339.

[41] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. 2015. Performance-influence models for highly configurable systems. In *10th Joint Meeting on Foundations of Software Engineering*. ACM, 284–294.

[42] César Soto-Valero. 2021. Software debloating papers. https://www.cesarsotovalero.net/software-debloating-papers.

[43] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* (2021), 1–44.

[44] David B Stewart. 2001. Measuring execution time and real-time performance. In *Embedded Systems Conference*, Vol. 141. Citeseer.

[45] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of software engineering research. *Transactions on SE and Methodology* 27, 3 (2018), 1–51.

[46] Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, and Olivier Barais. 2017. Learning contextual-variability models. *IEEE Software* (2017).

[47] Pavel Valov, Jianmei Guo, and Krzysztof Czarnecki. 2015. Empirical comparison of regression methods for variability-aware performance prediction. In *19th International Conference on Software Product Line*. ACM, 186–190.

[48] Alexander von Rhein, Thomas Thüm, Ina Schaefer, Jörg Liebig, and Sven Apel. 2016. Variability encoding: From compile-time to load-time variability. *Journal of Logical and Algebraic Methods in Programming* (2016), 125–145.

[49] Mark Weiser. 1984. Program slicing. *Transactions on Soft. Eng.* (1984), 352–357.

[50] Frank Wilcoxon, SK Katti, and Roberta A Wilcox. 1963. *Critical values and probability levels for the Wilcoxon rank sum test and the Wilcoxon signed rank test*. Vol. 1. American Cyanamid Pearl River (NY).

[51] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *10th Joint Meeting on Foundations of Software Engineering*. ACM, 307–319.