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

Xhevahire Tërnava, Mathieu Acher, Benoit Combemale

To cite this version:

Xhevahire Tërnava, Mathieu Acher, Benoit Combemale. Specialization of Run-time Configuration Space at Compile-time: An Exploratory Study. SAC 2023 - The 38th ACM/SIGAPP Symposium on Applied Computing, Mar 2023, Tallinn, Estonia. pp.1-10. hal-03916459

HAL Id: hal-03916459
https://hal.science/hal-03916459v1
Submitted on 30 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 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 de-bloat 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 de-bloating problem where a subset of run-time options to specialize are bloat of a configurable system. Several de-bloating approaches are proposed in the literature. Most of them provide a way for de-bloating 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, de-bloating 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 de-bloating 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

KEYWORDS

program specialization, performance, de-bloating, 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 de-bloat 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 de-bloating problem where a subset of run-time options to specialize are bloat of a configurable system. Several de-bloating approaches are proposed in the literature. Most of them provide a way for de-bloating 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, de-bloating 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 de-bloating 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

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.
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.

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, 17, 20, 24, 41, 46]), and has plenty of run-time options. We use the x264 at its recent commit db8d417 [3]. It has 114,475 lines of code (LoC) [3], 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.

\[
\text{./configure --disable-lsmash && make}
\]

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.

\[
\text{x264 --cabac --mbtree --mixed-refs -o vid.264 vid.y4m}
\]

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, We
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 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-secure, 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) To 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 file 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 analyze 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 quite light to learn, use, and make it possible to remove the enclosed code.

*Algorithm 1* shows the static approach that we conducted to de-define 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,
### Algorithm 1: Delimiting a Boolean run-time option

**Input:** compilable C-based system with run-time options  
**Output:** compilable C with delimited run-time option(s)  

```c
#pragma once

#define OPTION_YES 1
#define OPTION_NO 0

// Define boolean directives
#if OPTION_YES // option --cabac
  // code if the option is set
#elif OPTION_NO // option --no-cabac
  // code if the option is unset/disabled
#else
  // code if the option is neither set nor unset/disabled
#endif
```

1. For option ∈ \{specialized_options\} do  
2. for option = \{true, false\} do  
3. Without removeoption.h ∈ source_files in Θ  
4. #define OPTION_YES 1  
5. #define OPTION_NO 0  
6. Set OPTION_YES \→ \(O_{\text{specialized}} \in \Theta\)  
7. repeat  
8. Find code that implement its true value  
9. Enclose it within \#if OPTION_YES...\#endif  
10. until \(O_{\text{specialized}}\) is valid  
11. Set OPTION_YES  
12. Set OPTION_NO \(\rightarrow O_{\text{specialized}}\) is valid  
13. repeat  
14. Find code that implement its false value  
15. Enclose it within \#if OPTION_NO...\#endif  
16. until \(O_{\text{specialized}}\) is valid  
17. Set OPTION_NO  
18. end for  
19. end for
```

### 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.

- **Algorithm 1:** Delimiting a Boolean run-time option

### 4 EXPERIMENTAL DESIGN

#### 4.1 Hypothesis

We present three null hypotheses each to be supported or refuted by our analysis. \(H_0\) concerns the binary size, \(H_0\) the attack surface, and \(H_0\) the performance of a software system. Their respective alternatives are \(H_1\), \(H_2\), and \(H_3\).

- **\(H_0\):** The baseline software system and its specialization are not significantly different with respect to their binary size.
- **\(H_1\):** The specialized software system has smaller binary size than its baseline.
- **\(H_2\):** The baseline software system and its specialization are not significantly different with respect to their attack surface.

#### 4.2 Experimentation

First, the original and specialized program are used to encode eight carefully chosen videos (cf. Section 4.4) by the ten built-in presets 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 \(^5\). 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.

---

\(^5\)A preset is a set with run-time configuration options that are set at once.

\(^6\)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 | fast     | veryfast | fast | medium | slow | show | suspense | veryslow | placebo |
|--------------------------|-------------------|-----------|----------|----------|------|--------|------|------|----------|----------|---------|
| --no-mixed-refs         | S₁                | ×         | ×        |          |      |        |      |      |          |          |         |
| --no-psy                | S₂                | ×         | ×        |          |      |        |      |      |          |          |         |
| --no-cabac              | S₃                | ×         | ×        |          |      |        |      |      |          |          |         |
| --no-weightb            | S₄                | ×         | ×        |          |      |        |      |      |          |          |         |
| --no-mbtree             | S₅                | ×         | ×        |          |      |        |      |      |          |          |         |
| --no-cabac              | S₆                | ×         | ×        |          |      |        |      |      |          |          |         |
| --no-weightb            | S₇                | ×         | ×        |          |      |        |      |      |          |          |         |
| --no-mbtree             | S₈                | ×         | ×        |          |      |        |      |      |          |          |         |
| --no-cabac, --no-weightb| S₉                |          |          |          |      |        |      |      |          |          |         |

### Table 2: The properties of the eight used videos

| Name | Size [MB] | Length [sec] |
|------|-----------|--------------|
| V₁₁₂₇₂₀₀₆₈₀ | 108.4 | 13           |
| V₂₄₈₀x₃₆₀  | 155.6 | 20           |
| V₃₆₄₀x₅₆₀  | 160.5 | 19           |
| V₄₆₄₀x₅₆₀  | 172.5 | 19           |

### H₀₁: The specialized software system has an attacker smaller attack surface (i.e., # code reuse gadgets) than its baseline.

### H₀₂: The baseline software system and its specialization do not perform significantly different.

### H₀₃: The specialized software systems 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).

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 --no-mixed-refs, --cabac, --mbtree, --no-psy, and --no-weightb, 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.

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 remaining 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 example, the preset fast is specialized by removing one run-time option at a time. 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 $S₀$, for the baseline, and $S₁$ to $S₂₀$, 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 g++ 10.3.1 compiler to build the baseline and specialized systems of x264. Some of the done measurements (cf. $H₀₃$) require an input, namely a video. Instead of a random
5 RESULTS

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

5.1 The binary size of specialized system

To address H₀₁, 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 Scenario₁ and Scenario₂, measured their respective binary size, and compared them with the baseline binary size. Finally, we statistically test the H₀₁ 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 Scenario₂ (S₁₁ - S₂₀), as they are specialized by more than one run-time option, than those by Scenario₁ (S₁ - S₁₀). In percentage, compared to the baseline binary size, the specialized systems S₁₁ - S₂₀ have a reduced binary size between 0.001% and 5.416%, whereas S₁₁ - S₂₀ 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 α = 0.05 (t = 210, p = 9.54 · 10⁻⁷) ¹¹, therefore H₀₁ is rejected in favor of H₁₁. 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.

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 software tools and counted the overall gadgets in the system’s binary. Then, we statistically test the H₀₂ 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 S₁ - S₁₀ have fewer gadgets than the baseline system, between −0.42% and 6.60%, whereas specialized systems S₁₁ - S₂₀ 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 threat the system.

However, the calculated Wilcoxon signed-rank test shows that the H₀₂ is not rejected for S₁ - S₁₀ (t = 35, p = 0.25), whereas it is rejected for S₁₁ - S₂₀ (t = 55, p = 9.77 · 10⁻⁴). In general, it is rejected for S₁ - S₂₀ in favor of the H₀₂ (t = 190, p = 3.54 · 10⁻⁴). This suggests that it is better to specialize a software system by multiple run-time options (i.e., Scenario₂), 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, S₀ is specialized regarding the ‘-mixed-ref’ options (cf. Table 1). We should encode videos only with 4 presets in it, namely, ultrahigh, supreme, veryfast, and faster. The rest of the presets should not be used in S₀ as they require the option of ‘-mixed-ref’s, which

Figures 1: The binary size and the found number of gadgets in S₀ (baseline) and S₁ - S₂₀ (specializations). (mean ± std) for binary size - S₀: (3, 096, 176 ± 0), S₁-₂₀: (3, 020, 417 ± 63, 641), and for gadgets - S₀: (106, 495 ± 0), S₁-₂₀: (103, 414 ± 2, 696)

---

¹⁰UCG dataset: https://media.withyoutube.com/
¹¹Where t is the sum of the ranks of the differences above or below zero.

¹²ROPgadget tool: https://github.com/JonathanSalwan/ROPgadget
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 $S_0$. 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 Scenario1, Scenario2, 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 $S_0$ has no values, meaning that it has no available presets as all of them require the option of \texttt{--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 $S_{17}$ has the most improved encoding time, for 11.97%. This means that because of its removed unused options the preset fast in $S_{17}$ 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. $S_0$ 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 superb fast 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
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 $(t = 54, p = 1.95 \times 10^{-3})$ and frame rate $(t = 7, p = 0.02)$. In general, it is rejected for all specializations $S_1 - S_{20}$ in favor of $H_{03}$ 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,
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 configuration options 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, operational, 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 Jezequel. 2020. Sampling effect on performance prediction of configurable systems: A case study. In International Conference on Performance Engineering. ACM, 277–288.
Xhevahire Ternava, Mathieu Acher, and Benoit Combemale

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. Sys. and Software Product Line Conference. ACM, 46–57.

[29] Dylan McNamara, Jonathan Walpole, Calton Pu, Crispin Cowan, Christian Krätsch, Ashvin Goel, Perry Wagle, Carlos Consol, Gilles Muller, and Renaud Marlet. 2001. Specialization tools and techniques for systematic optimization of system software. Transactions on Computer Systems (2001), 217–251.

[30] Jens Meinicke, Chun-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 Newman, and Scott A. Watterson. 1997. TofoJava for applications-A way ahead of time (WAT) compiler. In COOTS ’97, 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] Vahidav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Simplifier: 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 Grebhan, Sven Acher, 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. Citeeseer.

[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, Jannei 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 Thiim, 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 Wessner. 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, Yuanpu 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.