Scratching the Surface of ./configure: Learning the Effects of Compile-Time Options on Binary Size and Gadgets

Xhevahire Tërnava, Mathieu Acher, Jean-Marc Jézéquel, Arnaud Blouin, Luc Lesoil

Scratching the Surface of `./configure`: Learning the Effects of Compile-Time Options on Binary Size and Gadgets

Xhevahire Tërnava[1], Mathieu Acher[12], Luc Lesoil[1], Arnaud Blouin[13], and Jean-Marc Jézéquel[1]

[1] Univ Rennes, CNRS, Inria, IRISA - UMR 6074, F-35000 Rennes
[2] Institut Universitaire de France (IUF)
[3] INSA Rennes
{firstname.lastname}@irisa.fr

Abstract. Numerous software systems are configurable through compile-time options and the widely used `./configure`. However, the combined effects of these options on binary's non-functional properties (size and attack surface) are often not documented, and or not well understood, even by experts. Our goal is to provide automated support for exploring and comprehending the configuration space (*a.k.a.,* surface) of compile-time options using statistical learning techniques. In this paper, we perform an empirical study on four C-based configurable systems. We measure the variation of binary size and attack surface (by quantifying the number of code reuse gadgets) in over 400 compile-time configurations of a subject system. We then apply statistical learning techniques on top of our build infrastructure to identify how compile-time options relate to non-functional properties. Our results show that, by changing the default configuration, the system's binary size and gadgets vary greatly (roughly -79% to 244% and -77% to 30%, respectively). Then, we found out that identifying the most influential options can be accurately learned with a small training set, while their relative importance varies across size and attack surface for the same system. Practitioners can use our approach and artifacts to explore the effects of compile-time options in order to take informed decisions when configuring a system with `./configure`.

Keywords: Configurable systems, compile-time variability, binary size, gadgets, system security, non-funcional properties, statistical learning

1 Introduction

Modern software systems are highly configurable and expose to the users their abundant configuration options. By enabling or disabling configuration options, a software system can be customized for different contexts, such as for different users or hardware with a limited memory size, without the need to modify its source code. But this high flexibility of software systems comes with a cost.

A large number of configuration options indeed makes the system complex and threatens its maintenance. First, because *"a significant percentage (up to* 54.1*%) of parameters are rarely set by any user."* [28] and thus they unnecessarily bloat the software [14]. Then, the disabled options may have security issues or bugs, which might be exploited and threaten the whole system. Further, there is evidence that many software failures arise from misconfiguration [19,29], while finding the cause of a failure among the large set of options is difficult. To reduce the complexity and improve the software configuration quality, there are several approaches [28,21,3,15]. But the large number of configuration options and their enabling/disabling are likely to have an impact also on the non-functional properties of a given system, such as in its binary size and attack surface.

Actually, most of embedded systems, mobile devices, or any resource-constrained devices may exhibit requirements regarding the executable binary size of a software application [11]. Further, from a security perspective, a large code base of any configurable software system increases the possibilities of *gadgets*, that is, of small code chunks that an attacker can chain in order to build an exploit [4,21,3]. Therefore, motivated by concrete requirements in real software systems (*cf.* Section 2), it is of great importance to explore the effects of configuration options on these two non-functional properties of a given system. There are approaches on measuring a set of non-functional properties, including the binary size, for a derived software system in the context of software product lines [26,25]. But, there is hardly any work on exploring the compile-time configuration space during a system's build with `./configure`, which is extensive on C-based software systems. In particular, we are unaware of works that attempt to measure *(i)* how much vary the executable binary size and gadgets of a system based on its compile-time configuration options, *(ii)* whether some options are more influential than the others, and *(iii)* whether there is a relationship between these two non-functional properties changes. By making explicit these effects, a user may easily find the unneeded and most vulnerable options in order to reach a desired executable binary size, to prevent any code reuse attack, to improve the installation speed, or the occupied memory size of a system on a device.

The contributions of this paper are as follows:

- We provide empirical evidence for the great variation of binary size and attack surface in C-based systems, depending on their applied compile-time configurations. We argue that changing the system's default configuration can be beneficial for its users, but how it should be changed is not trivial.
- Therefore, we made a comparison of different learning techniques to predict binary size and gadget of any compile-time configuration. We then report and qualitatively analyse the influential options and their interactions based on interpretable information of performance prediction models.
- Furthermore, we provide the dataset of our measurements, as well as our scripts, which can be used to reproduce our study.

To accomplish them, we provide the motivation (Section 2) and research questions to be addressed (Section 3). Then, we set up an experiment to answer

those questions by studying four popular C-based open-source systems (Section 4). Next, we report on the variation of binary size and number of gadgets, measured on a large set of configurations (Sections 5.1 and 5.2). In addition, we provide an approach to find the most influential options on these two non-functional properties and report on them for four systems (Section 5.3). We also discuss the costs, benefits, and future work (Section 6), including threats to validity (Section 7) and related work (Section 8). Section 9 concludes our paper.

2 Background and motivation

In this section, we provide a background on the `./configure` flavour for configuring C-based systems and on the *number of gadgets*, as an attack surface metric. At the same time, the motivation of our study is presented.

`./configure`. Most of the C-based software systems, which are also the subjects of this work, use the GNU autotools (*e.g.,* Autoconf [5]) to help developers to pack and distribute their software to multiple platforms and to facilitate their configuration and installation by the end-users. All that end-users see is the packed software with the generated *configure* script and *Makefile.in* file. Then, all that is left for the users to do is simply to type the command sequence `./configure && make && make install` in order to configure, build, and install the given software. As most of the C-based systems are configurable through compile-time and run-time options, users can use the `./configure` flavour to customize the targeted software at compile-time. For example, `x264` is a video encoder[4] with 39 compile-time options. In case the support for mp4 video encoding is not required, then it is possible to deactivate it by using the `--disable-lsmash` option during the system build, as in the following listing.

```
$ ./configure --disable-lsmash  # It generates Makefile from Makefile.in
$ make                          # It uses Makefile to build the x264
$ make install                  # It uses Makefile to install the x264
```

In this way, the `./configure` makes it possible to customize a software system with only the needed functionalities. Despite this possibility, a software system is often installed with off-the-shelf default configuration options. But, there is evidence that system administrators frequently make poor configuration choices, for example, the default settings for Hadoop result in the worst possible performance [9]. Hence, system administrators often suggest users to customize software systems to get their desired systems' performance or non-functional properties. Specifically, excluding certain unused functionalities at compile-time is often seen as an opportunity to reduce the system's binary size for cases where it is an important factor, such as in embedded applications. Such an example is `SQLite`, stating that *"If optional features are omitted, the size of the `SQLite` library can be reduced below 180KiB."* [5] for which reason `SQLite` is also a popular

[4] x264 settings: http://www.chaneru.com/Roku/HLS/X264_Settings.htm
[5] `SQLite`: http://barbra-coco.dyndns.org/sqlite/about.html

database engine in memory-constrained devices. But, unlike system administrators, end-users lack the expertise to tune the system to get the right binary size. Basically, they lack knowledge on: Which are the optional compile-time options in a system that should be removed to get a desired binary size? What is the effect of each enabled or disabled option on the system's binary size?

Gadgets. Nowadays, the security of modern software systems is mostly threatened internally, that is, by reusing their existing code, without the need for code injection [21]. 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 [4,3,21]. Hence, to a certain degree, the attack surface of systems is related to their binary size. Therefore, considering that the security in software systems is important, such as in `SQLite`[6], as few gadgets (smaller binary size) are often desirable to reduce the attack surface in the system. But, end-users lack the knowledge regarding how much the unused compile-time options in their installed system with a default configuration threaten their system? Or, what is the effect of each compile-time option on the system's attack surface?

Though numerous works have considered the performance of software product lines [26,25], little is known about the effects of compile-time options on the system's non-functional properties, namely, on binary size, number of gadgets, and how configuration knowledge related to binary size or attack surface can be effectively recovered. The need to make users aware of the importance of customizing software during `./configure` and system administrators to document the effects of options on binary size and gadgets motivates our work.

3 Research questions

Motivated by such examples, the goal of this study is to quantify and learn the effects of compile-time options on binary size and attack surface of C-based configurable software systems and which options are of great importance for end-users. To attain this goal, we define the three following research questions.

- RQ_1: **What is the effect of compile-time options of a system on its binary size?** To this end, we use four C-based configurable software systems, as subjects, and enable or disable their compile-time options based on two scenarios (*cf.* Section 4.3). We then record the binary size of each obtained system's executable and compare them with the baseline binary size.
- RQ_2: **What is the effect of compile-time options of a system on its attack surface (*a.k.a.*, gadgets)?** To measure the attack surface of a

[6] Security in `SQLite`: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=sqlite

Table 1. Four subject systems with their baseline binary size [bytes] and gadgets

Subject system	Description	Analysed commit	#LoC	#Options	Baseline Binary size	Baseline Gadgets
`x264`	*Video encoder*	db0d417	114,475	25/39	3,096,112	106,878
`nginx`	*Web server*	cc73d76	172,368	91/127	4,507,168	29,925
`SQLite`	*SQL database engine*	385b982	318,496	31/72	8,561,208	67,734
`xz`	*Data compressor*	e7da44d	37,489	36/88	1,254,536	9,121

system, we rely on the number of found gadgets in its executable. In addition, we explore whether there is a correlation between the binary size and the number of gadgets for different enabled/disabled compile-time options.

RQ_3: **Which compile-time options are the most influential on the binary size and the gadgets of a software system?** To this end, we use statistical prediction models to predict the influence of compile-time options on the binary size and number of gadgets in our subject systems. We report on interactions among options and discuss qualitatively these options.

4 Experimental Protocol

In this section, we introduce the used subject systems and the experiment settings in order to answer the three research questions.

4.1 Subject systems

To select a subject system for our study, we used several sources, such as the studied systems by research papers on software variability and software debloating[7], the website openbenchmarking[8], and our knowledge on popular open-source projects. Then, we had into consideration the fact that subjects should contain compile-time configuration options, are open-source, cover different application domains, and are popular projects. To reason on a project's popularity, we used as a proxy the number of stars, commits, and contributors in its git repository.

As the C-based software systems are typical systems that are rich with compile-time configuration options to be handled, we selected four of them as subjects, namely, `x264`, `nginx`, `SQLite`, and `xz`. Regarding their popularity, all of them have between 86–15.6k stars, 1.3k–23.7k commits, and 17–94 contributors[9].

In Table 1 are given a brief description of each subject system, its respective analysed commit ID in its git repository, size in number of Lines of Code (LoC)[10], and the considered number (versus the overall number) of compile-time options.

[7] https://www.cesarsotovalero.net/software-debloating-papers#2020
[8] https://openbenchmarking.org/
[9] Based on our last check on February 2022
[10] Measured using the `cloc` tool: https://github.com/AlDanial/cloc

4.2 Baseline configuration

Usually, by using the `./configure --help` option during the build of a C-based system, the available compile-time options are shown as a plain list, including the external libraries and their default values. These are the used options to customize the given system for a specific environment and user context. It is important to note that, each system comes with a default build configuration, that is, each of its compile-time options is by default either enabled or disabled. We refer to this default configuration of a system as its *baseline configuration*.

The baseline configuration of our subject systems is the current configuration in their respective git repository (*cf.* Table 1). In order to exercise with as many compile-time options as possible, we had to install in our environment the external libraries required by each subject. Furthermore, Table 1 presents the executable binary size in bytes and the number of found gadgets[11] for the baseline configuration of each subject system. In the following, we will refer to them as the *baseline binary size* and *baseline number of gadgets*.

4.3 The conducted experiment

With all four subject systems, we conducted an experiment in the same environment. Specifically, we first fetched the targeted system from its git repository and compiled it on its default configuration. To make sure that the compilation was successful, we used the compiled system in an elementary example. For instance, we encoded a video using `x264`, used a run-time option of `nginx`, opened a database using `SQLite`, and compressed a file using `xz`. To later automate the generation of configurations, we manually explored the configuration space of each system, identified the dependencies of its compile-time options, and build its feature model (FM)[12] (their availability is given below).

Next, to answer the research questions RQ_1 and RQ_2, we automatically customized the four subject systems by following these two scenarios.

S_1 : First, each system is customized by **a single compile-time option** at a time. The left value in column `#Options` in Table 1 shows the number of considered options in each system. It has to be stressed that these options had Boolean or enumerate values. Hence, using them, we built 31 configurations with a single option for `x264`, 91 for `nginx`, 31 for `SQLite`, and 65 for `xz`.

S_2 : Then, each system is customized by **a mixed set of compile-time options**. In total, we used a significant sample of 400 configurations in each subject system. All of these configurations are generated using the random product generator in the FeatureIDE framework.

Finally, to answer RQ_3, we designed the following protocol to identify *influential* compile-time options *i.e.,* options having a statistically significant influence on non-functional properties. There exist several techniques to identify

[11] Measured using the ROPgadget: https://github.com/JonathanSalwan/ROPgadget
[12] For this purpose, we used FeatureIDE framework: https://featureide.github.io/

influential options, most of them based on supervised machine learning models[13] predicting the performance properties of our systems. We first rely on the measurement of *feature permutation importance* [16] (in short: feature importance). Feature importance is computed through the observation of the effect on machine learning model accuracy of randomly shuffling each predictor variable. We compute feature importance over random forest [16], the machine learning method leading to the best accuracy in our case[14]. Feature importance gives a score between 0 (no influence) and 1. It is comparable across different problems, *e.g.,* we can compare the influence of the same compile-time option over binary size and gadgets. The measure automatically takes into account all interactions with other features. This is a good property but also a disadvantage since feature interactions are not made explicit. To further understand and mitigate this lack, we consider: *(i)* the coefficients of Lasso [8] with feature interactions for gadgets; and *(ii)* the rules of the decision trees that give information on how features interact. We then confront identified options with the documentation of the project in order to understand whether their effect on size and gadgets make sense from a domain or technical point of view.

Moreover, all steps of our experiment are automated by Python scripts. The used artifacts, such as the Feature Model and over 400 selected configurations of each system, the details to reproduce our experiment, and all the obtained results are made available in the git repository https://github.com/diverse-project/confsurface and in zenodo https://doi.org/10.5281/zenodo.6401250.

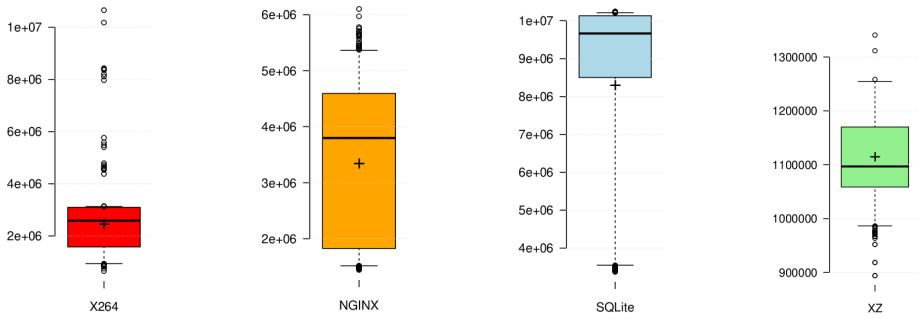

Fig. 1. The variation of binary size in four soubject systems

[13] Owing to space issue, we place the technical details about our machine learning models in the companion repository and encourage our readers to consult them; the implementation, the chosen learning methods, the choice of metric and the obtained results when predicting the binary size and the number of gadgets of our systems.

[14] See the detailed results at https://github.com/diverse-project/confsurface/blob/main/learning/results.md

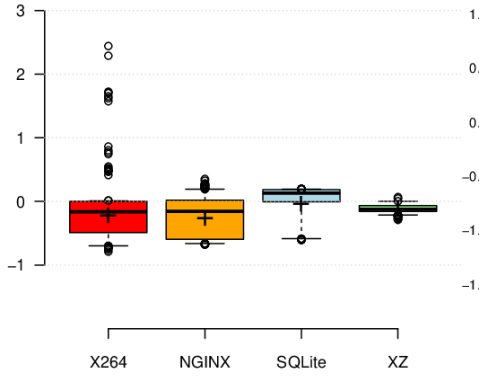

Fig. 2. Binary size baseline comparisons

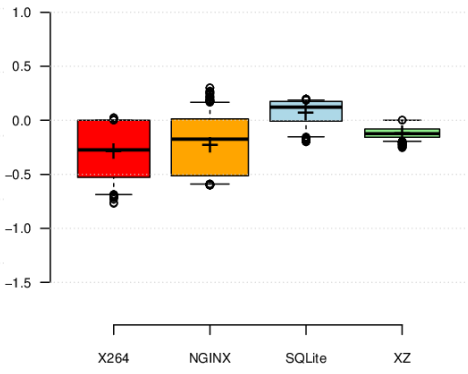

Fig. 3. Gadgets baseline comparisons

5 Results

We now report the results and observations with regard to our research questions.

5.1 The effect of compile-time options on binary size (RQ_1)

By using over 400 sample configurations based on two given scenarios in Section 4.3, we measured the binary size of each subject after its customization.

Figure 1 shows resulting variation of the binary size of each subject system. Depending on the used configuration, we found out that the binary size in all four systems varies considerably. Namely, in `x264` between 0.62 MiB and 10.16 MiB, in `nginx` between 1.38 MiB and 5.82 MiB, in `SQLite` between 3.22 MiB and 9.77 MiB, and in `xz` between 0.85 MiB and 1.28 MiB. The colored part of box plots of `x264`, `SQLite`, and `xz` suggests that these systems hold quite similar binary sizes for roughly 50% of their configurations. But, the far upper outliers signify that there are some configurations in `x264`, `nginx`, and `xz` that lead to a far higher binary size. Still, the bottom outliers suggest that `x264`, `SQLite`, and `xz` can reach a far smaller binary size in a considerable number of configurations. Figure 2 shows the relative difference, in percentage, of a system's binary size after its customization to the baseline binary size. By comparing with the *baseline binary size* given in Table 1, it can be observed that most of the configurations in three of the four systems lead to a smaller binary size than their baseline binary size (0 percentage in Figure 2 is the baseline value). Specifically, 56% of configurations in `x264`, 68% in `nginx`, 28% in `SQLite`, and 92% in `xz` provide a smaller binary size. Fewer configurations increase the system's binary size, namely, 6% in `x264`, 32% in `nginx`, 67% in `SQLite`, and 1% in `xz`. Then, 38%, 5%, and 7% of the configurations in `x264`, `SQLite`, and `xz`, respectively, have the same binary size as in their default configuration. Only in `nginx`, for any other configuration that is different from its baseline configuration, its binary size is always different.

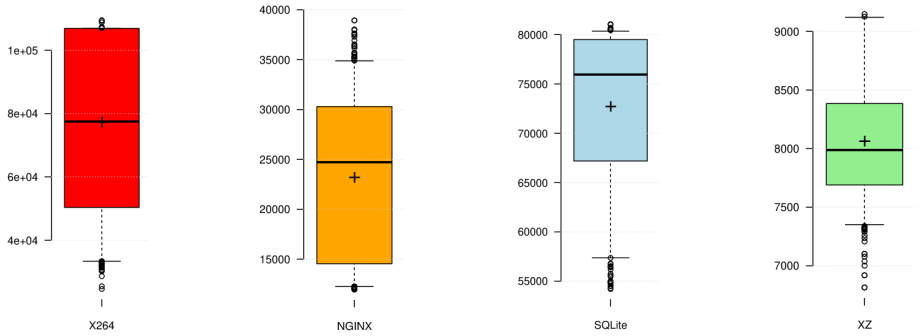

Fig. 4. The variation of gadgets in four subject systems

Answer to RQ_1***:*** These results show that compile-time options of a system have a noticeable effect on its executable binary size. Based on four popular systems, changing the system's default configuration, it is very likely (in 61% of the cases) that the system's binary size will be reduced, but it is less likely (in 26% of the cases) that it will be increased or remained the same (in 13% of the cases). Then, the binary size on average decreases for more bytes (36%) than that it increases (33%). Hence, the compile-time customization of a software has mainly a positive effect on its binary size.

5.2 The effect of compile-time options on attack surface (*RQ*$_2$)

Using the same sample of configurations, S_1 and S_2 in Section 4.3, we measured the number of code reuse gadgets of each system after its customization.

The variation of the number of gadgets. Figure 4 shows resulting variation of the number of gadgets in each subject system. By changing the baseline configuration of a system, we found out that the number of gadgets in all four systems varies considerably. Specifically, in `x264` between 25K and 109K, in `nginx` between 12K and 39K, in `SQLite` between 54K and 81K, and in `xz` between 7K and 9K gadgets. The tall box plots of four systems suggest that each system configuration provides a customized system with a quite different number of gadgets. Figure 3 shows the relative difference, in percentage, of a system's gadgets after its customization to the baseline gadgets. By comparing with the *baseline number of gadgets* given in Table 1, it can be noticed that most of the configurations in `x264`, `nginx`, and `xz` (61%, 68%, and 92%, respectively) lead to a smaller number of gadgets than the baseline configuration (0 percentage in Figure 3 is the baseline value). Whereas, only 30% of them are in `SQLite`. Still, only in a few configurations the number of gadgets is increased (1% in `x264`, 32% in `nginx`, and 0.43% in `xz`) or remained the same (38% in `x264`, 4% in `SQLite`, and 7% in `xz`) with the baseline number of gadgets. As with its binary size (*cf.* Section 5.1), the number of gadgets in `nginx` is always different for any other configuration that is different from its baseline configuration.

The correlation between binary size and gadgets. The shown results in RQ_1 and RQ_2 suggest that the attack surface (*i.e.,* gadgets) and binary size may correlate. To prove if this is the case, we compute the Pearson correlation coefficient for each subject system. It is a widely used measure of linear correlation between two distributions. The extreme values of -1 and 1 indicate a perfectly linear relationship, whereas a coefficient of 0 represents no linear relationship. In addition, we also report on Spearman rank correlation [12]. A value of 1 indicates a similar rank of configurations (*e.g.,* roughly, the configurations leading to smaller binaries remain the same as the configurations leading to fewer gadgets).

As a result, we found out that `nginx` has almost a perfect correlation of 0.99 (in both Pearson and Spearman correlation). And, `SQLite` has a very strong correlation, with Pearson being 0.90 and Spearman 0.98. This signifies that configurations in `nginx` and `SQLite` have the same effect on their binary size as in their number of gadgets. Next, `xz` has a very strong correlation, but not perfect, with Pearson being 0.85 and Spearman 0.83. This suggests that there are configurations that have a different effect on the binary size from that in the number of gadgets. On the other hand, `x264` differs from the three other systems. It has a weak to medium, positive, correlation, with Pearson being 0.52 and Spearman 0.82. In this system, we noticed that there are several options in its configurations that increase its binary size but reduce the number of its gadgets.

Answer to RQ_2***:*** These results show that the attack surface of a software system highly depends on its used compile-time options during its build. Based on four popular systems, changing the system's default configuration, it is more likely that its number of gadgets will get reduced (in 63% of the cases) than they will get increased (in 25% of the cases) or remain the same (in 12% of the cases). On average, by enabling/disabling new options, the attack surface will be reduced far more (25%) than that it will be increased (6%). Moreover, there is a weak (0.52) to almost perfect (0.99) Pearson correlation between the variation of binary size and the number of gadgets in a given system.

5.3 Influential compile-time options (RQ_3)

Which options are the most influential in these systems? To answer it, we did a detailed analysis of the effect of each compile-time option on the system's binary size and gadgets, which is described in Section 4.3. The results are as follows.

`x264`: As shown in Figure 5 (right), `--system-libx264` is by far the most important option for predicting gadgets in `x264` (around 80% of the importance). However, the prediction of binary size involves much more options and interactions among `--enable-debug`, `--enable-strip`, `--disable-lsmash`, and `--disable-asm`. That is, there are more influential options (and more interactions to capture through learning) for size than for gadgets. It partly explains why achieving low prediction errors with gadgets is easier than with size (further details are given in the companion web page). The options' effects we have

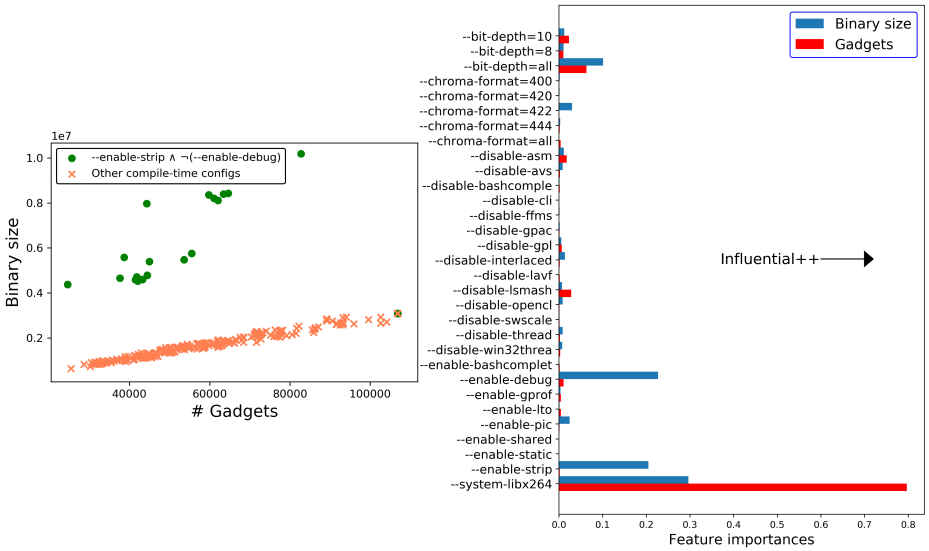

Fig. 5. An interaction example of two options in `x264` that changes its binary size but not its number of gadgets (left). Feature importance in `x264` (right)

learned also make sense: identified options are related to the compiler behavior or libraries linked to the binary.

nginx: The option `--without-http` is the most influential (feature importance greater than 0.85). It is quite intuitive since this option deactivates a major functionality. It should be noted that `--without-http` corresponds to a realistic usage, since `nginx` can be used as *e.g.,* a reverse proxy or to handle arbitrary TCP connections. However, the strong importance of `--without-http` tends to hide the fact that other options are actually influential w.r.t. size and gadgets. Looking at the decision tree[15], we can observe that many options are actually interacting with the option `--without-http`. Such interactions are actually needed to reach high accuracy. Our learning can retrieve options like `--without-http_proxy_module`, `--with-stream`, *etc.* An expert can certainly intuit the positive or negative effect, but neither to quantify the strength of the effects nor to capture interactions with other configuration choices. Hence, an expert can rely on the prediction model of `nginx` to explore configuration tradeoffs between functionality, size, and attack surface.

SQLite: Its most influential options are by far `--enable-all` and `--enable-fts5` with more than 90% of the importance. The importance of the first option is intuitive and its effect on binary size or gadgets is obvious since numerous features are added. The other option *"FTS5 is currently disabled by default for the*

[15] See https://github.com/diverse-project/confsurface/tree/main/nginx/nginx.pdf

source-tree configure script", but it *"is included as part of the* `SQLite` *amalgamation"*[16]. There are two other options `--enable-geopoly` and `--enable-session` that have an influence, but we observe very few interactions for binary size or gadgets. In summary, the *"take it all"* feature and the identification of an individual option (fts5) is sufficient to accurately configure `SQLite` w.r.t. size and gadgets.

xz: Compared to `SQLite`, the importance is more spread among compile-time options for `xz`. The maximum is 0.17 for binary size and 0.24 for gadgets, and there are 6 main influential options. We also observed many pair-wise interactions (*e.g.,* `--enable-sandbox` with `--enable-small`).

Are the influential options of a system the same for binary size and gadgets? We observed that the ways how options have an effect on the `x264`'s binary size and gadgets differ, as shown in Figure 5 (right). For instance, `--system-libx264` has a strong importance for gadgets (0.78), but its importance is much lower (0.18) for binary size. The intent of this option is to *"use system libx264 instead of internal"*. Hence, an interesting result of our learning process is that the way a code is integrated may have a different impact on attack surface and binary size. For instance, `--disable-lsmash` has almost no effect on gadgets but plays a key role in size prediction. Similarly, `--enable-debug` and `--enable-strip` options have negligible influence on gadgets while their importance for size is 0.19 and 0.17, respectively. Figure 5 (left) suggests that a combination of `--enable-strip` set to `True` and `--enable-debug` set to `False` for a compile-time configuration increases drastically the associated binary size of `x264` (+122.6% of binary size, on average, compared to the other compile-time options).

For `nginx` and `SQLite`, the influential options are exactly the same; it is not surprising owing to the strong correlation. However, it is good news for the reuse of prediction model and configuration knowledge. As for `xz`, however, the option `--enable-checks` is the most important for binary size (0.17), but not for gadgets (0.05). On the other hand, `--enable-small` is the most important option for gadgets (0.24) but not for size (0.05). We have carefully verified, there is no dependency and collinearity between these two options. Hence, the "switch" between `--enable-small` and `--enable-checks` mainly explains the strong but not perfect correlation between binary sizes and gadgets in `xz`. Besides, `--enable-checks` has a strong influence on the `xz`'s binary size, but disable integrity checks and the documentation[17] warns that *"this option should be used only when it is known to not cause problems"*. All these observations tend to show the complexity of configuring a system w.r.t. size, gadgets, and functionality.

Answer to RQ_3***:*** Our learning process can identify influential options that make sense from a domain knowledge point of view. Moreover, our prediction model can be used to take informed decisions when customizing a software system

[16] According to https://sqlite.org/fts5.html, last access February 2022
[17] https://github.com/xz-mirror/xz/blob/master/INSTALL, last access February 2022

at compile-time. We have also shown that options' effects and interactions on either binary size or gadgets can vary. As a user, it makes the tuning of software systems difficult to achieve at compile-time.

6 Discussion

In this section, we discuss our findings and share insights about the costs and benefits of learning the effects of compile-time configurations.

Costs. Findings from RQ_1, RQ_2, and RQ_3 show that knowing the precise effects of compile-time options is possible, but comes at a price. We noticed that there is a triple cost:

1. There is a *human cost* to automate the generation of a well-built system configuration. A configuration with a wrong combination of options, that have dependencies, usually triggers a warning or error during the system's build. Based on four systems, the options' dependencies are hardly documented. That is why we retrieved them manually when we build the feature models, requiring several tries and fixes. In addition, the provision of reusable containers with pre-installed libraries and tools (*e.g.,* Dockerfiles) can decrease the burden of developers, users, and researchers in charge of the building.
2. Then, to quantify the effects of each compile-time option on binary size and gadgets, but not only, the system needs to be built each time. But, this can be costly in terms of the *required time and disk resources*. For instance, the time to build `x264` and `NodeJS` (another configurable system) in their default configuration varies between a few seconds and 30 minutes, respectively.
3. Lastly, there is also a smaller but important cost, the *learning cost*. Once the effects of options are measured and quantified, it is possible to instrument the different machine learning methods in order to obtain the final list of influential options and thus predict the best compile-time options to use w.r.t. user constraints. However, a tradeoff should be found between the accuracy of the learning and the time needed to train the models. For instance, in our case, *(i) random forest* is the most accurate and more stable, reaching low prediction errors with a relatively small number of compilations, *(ii) decision tree* is also competitive, less accurate but faster to train, whereas *(iii) linear regression* is both unstable and inaccurate[18].

Benefits. Despite these costs, having information about the effects of compile-time configuration space on the non-functional properties of a system is beneficial for different stakeholders. First, developers can use that knowledge to build and test only configurations with resolved dependencies and to find the best baseline configuration for users for which the system's binary size or security matters. It should be noted that the configuration knowledge about the binary size and

[18] Detailed results about the tradeoff cost-accuracy of machine learning models can be consulted at https://github.com/diverse-project/confsurface/tree/main/learning/results.md

surface attack is poorly documented in the four projects. This knowledge is also non-trivial: basic linear regression models are poorly accurate since interactions are not taken into account. In response, we are capable of synthesizing accurate information that is both interpretable and actionable to quantify the effects of options and their interactions. Developers can better document their projects and provide an infrastructure capable of building any configuration. Second, the identification of influential options (and interactions) will help users to have a quicker intuition on how to customize a given system in order to reach their purpose. Users can predict the properties of configurations without actually building them. An open issue is how to build tools (*e.g.,* configurators) on top of feature models and inferred knowledge to further assist users when configuring the compilation of their systems. In particular, we provided evidence that binary size and surface attack are not necessarily correlated and possibly conflicting. Beyond security and size concerns, there are other non-functional properties (*e.g.,* execution time) to consider, calling to explore tradeoffs and resolve multi-objective problems with automated guidance. We leave it as future work.

7 Threats to validity

Internal validity. A first internal threat stems from the installed external libraries in our environment, which resemble an instrumentation threat. In this study, we do not report on the version of the installed libraries in our experimentation environment, which are required by subject systems. We always installed the last possible version of each library. But, if our experiment is reproduced by installing an older or newer version of them then the resulting binary size, gadgets, and the effects of options may differ. For this reason, providing a Docker image to precisely reproduce our experiments is envisioned in our future work. It can also be useful for developers and maintainers. Another instrumentation threat is related to the manual identification of dependencies between compile-time options during the build of feature models for each subject system. New or other dependencies between compile-time options of the considered systems can be present, still, for all over 400 considered configurations per system, we make sure that the system is at least always compilable. Hence, on our next future step is to automatically identify the dependencies between compile-time options and to build the feature model of a given software system. A further internal threat is related to the sample of measurements used to test our prediction models. The sample may not be representative of the whole configuration space, which may threaten the supposed qualities of the models. To mitigate this threat, we use random sampling and compile hundreds of configurations. We also notice that the accuracy of learning models tends to reach a plateau with the increase of the training with no overfitting – it is a good signal.

External validity. While we believe that the four selected subject systems from different application domains show the effect of compile-time options in two non-functional properties of typical configurable software systems, still, we considered

only the C-based systems. Therefore, the considered number and used language of subject systems do not enable us to conclude that the results of a newly added system will always be in the same range as our obtained results.

8 Related Work

There are numerous works about the non-functional properties and performance of configurable systems (*e.g.,* see [13,18,10,23,6,24,27]). The idea is to build prediction models out of a sample of measurements. Non-functional properties are usually runtime performance (*e.g.,* execution time, memory consumption) that require the actual execution of the compiled system. In this work, we have considered non-functional properties that can be computed at compile-time, such as binary size and gadgets. Footprint has been subject to attention in [24] for Java-based and C-based systems. We have focused our effort on C-based projects with the `./configure` facility. Our goal was to instrument a build infrastructure capable of compiling any configuration, making no assumptions about which options to consider or not. To the best of our knowledge, the effects of compile-time options on attack surface (gadgets) have not been considered in this context.

Halin *et al.* [7] built a testing scaffold for the entire configuration space of JHipster, a Web generator. They reported that building a variability-aware testing infrastructure requires a substantial engineering effort to cover all design, implementation and validation activities. We have shared similar difficulties for elaborating the FMs and anticipating all possible libraries required by any compile-time configuration (see Section 6). Our work can be seen as a re-engineering effort to make configurable `./configure`. This is also an important topic in software product line engineering. Many techniques have been proposed to locate features, synthesize FMs out of artifacts, or recover an architecture out of variants [1,20,22,2,31]. In our study, we have started with a manual approach when recovering the informal documentation. Automated techniques come after to validate and refine both the FM and the build infrastructure.

There are several works in highly configurable systems that analyse the build files, such as Makefiles, of a system in order to extract its configuration or variability knowledge [17,30]. The main reason of these approaches is to detect the variability anomalies, such as dead code, that steams from Makefiles or during the build time of the system.

Xu *et al.* analyse over 600 real-world configuration issues in 4 subject systems to understand the consequences of too many configuration options (*a.k.a.,* knobs) [28]. They propose a way to simplify the configuration space of a system by removing, hiding, or categorizing them. Unlike them, we propose to configure a system by taking into consideration its binary size and attack surface.

9 Conclusion

Modern software systems are highly customizable through the compile-time options, which are especially extensive in C-based systems. These options are en-

abled or disabled during a system's build through the widely used `./configure`. While they have an evident impact on the functional properties of a system, their effect on its non-functional properties is hardly explored. In this work, we investigate the effect of compile-time options on the binary size and attack surface of a system by using four C-based systems. Our obtained results show that:

- Depending on the used compile-time options in a configuration, the binary size and number of gadgets can be increased (244.13% and 30.08%, respectively) or decreased considerably (78.98% and 76.97%, respectively), compared to the baseline system configuration. Whereas, the variation of gadgets has a weak (0.52) to almost perfect (0.99) correlation to the binary size of a system.
- Then, we show that the interactions among compile-time options can be best captured by expressive learning models. Our build infrastructure and learning process can accurately find the most influential options for binary size or gadgets. Our results show that developers and integrators can use prediction models to take informed decisions when configuring a system.

In short, practitioners can benefit from configuration knowledge that is non-trivial to quantify and otherwise undocumented. We will consider integrating configuration tools into these software projects to achieve size and attack surface goals, possibly with other (conflicting) functional or performance concerns. One lesson learned is that the computational and engineering cost of automating the exploration of the configuration space is not negligible and may be a barrier to the adoption of our approach for existing configurable projects. Modelling, reverse engineering, and learning techniques to assist developers in "scratching the surface" are therefore welcome. As future work, we plan to extend the study with more subjects, also implemented in other languages, and further consider compile-time options including compiler flags.

Acknowledgements. This research was funded by the SLIMFAST with DGA-Pôle Cyber (PEC) and Brittany region and the ANR-17-CE25-0010-01 VaryVary projects.

References

1. Assunção, W.K., Lopez-Herrejon, R.E., Linsbauer, L., Vergilio, S.R., Egyed, A.: Reengineering legacy applications into software product lines: a systematic mapping. Empirical Software Engineering **22**(6), 2972–3016 (2017). https://doi.org/10.1007/s10664-017-9499-z
2. Bécan, G., Acher, M., Baudry, B., Nasr, S.B.: Breathing ontological knowledge into feature model synthesis: An empirical study. Empirical Software Engineering **21**(4), 1794–1841 (2016). https://doi.org/10.1007/s10664-014-9357-1
3. Brown, M.D., Pande, S.: CARVE: Practical security-focused software debloating using simple feature set mappings. In: Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation. pp. 1–7 (2019). https://doi.org/10.1145/3338502.3359764

4. Brown, M.D., Pande, S.: 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 (CSET 19) (2019), https://www.usenix.org/system/files/cset19-paper_brown.pdf
5. GNU: Autoconf - GNU Project, https://www.gnu.org/software/autoconf/
6. Guo, J., Czarnecki, K., Apel, S., Siegmund, N., Wąsowski, A.: Variability-aware performance prediction: A statistical learning approach. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 301–311. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693089
7. Halin, A., Nuttinck, A., Acher, M., Devroey, X., Perrouin, G., Baudry, B.: Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. Empirical Software Engineering **24**(2), 674–717 (2019). https://doi.org/10.1007/s10664-018-9635-4
8. Hampel, F.R., Ronchetti, E.M., Rousseeuw, P.J., Stahel, W.A.: Robust statistics: The approach based on influence functions, vol. 196. John Wiley & Sons (2011)
9. Herodotou, H., Lim, H., Luo, G., Borisov, N., Dong, L., Cetin, F.B., Babu, S.: Starfish: A self-tuning system for big data analytics. In: Cidr. vol. 11, pp. 261–272 (2011)
10. Jamshidi, P., Siegmund, N., Velez, M., Kästner, C., Patel, A., Agarwal, Y.: Transfer learning for performance modeling of configurable systems: An exploratory analysis. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 497–508. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115661
11. Jiang, Y., Bao, Q., Wang, S., Liu, X., Wu, D.: RedDroid: Android application redundancy customization based on static analysis. In: 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE). pp. 189–199. IEEE (2018). https://doi.org/10.1109/ISSRE.2018.00029
12. Kendall, M.G.: Rank correlation methods. Harvard Book, Harvard (1948)
13. Lesoil, L., Acher, M., Tërnava, Xh., Blouin, A., Jézéquel, J.M.: The interplay of compile-time and run-time options for performance prediction. In: Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A. pp. 100–111 (2021). https://doi.org/10.1145/3461001.3471149
14. McGrenere, J., Moore, G.: Are we all in the same" bloat"? In: Proceedings of the Graphics Interface 2000 Conference, May 15-17, 2000, Montr'eal, Qu'ebec, Canada. pp. 187–196 (May 2000). https://doi.org/10.20380/GI2000.25
15. Meinicke, J., Wong, C.P., Vasilescu, B., Kästner, C.: Exploring differences and commonalities between feature flags and configuration options. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice. pp. 233–242 (2020). https://doi.org/10.1145/3377813.3381366
16. Molnar, C.: Interpretable Machine Learning. Lulu.com (2020)
17. Nadi, S., Holt, R.: The Linux kernel: A case study of build system variability. Journal of Software: Evolution and Process **26**(8), 730–746 (2014), https://doi.org/10.1002/smr.1595
18. Pereira, J.A., Acher, M., Martin, H., Jézéquel, J.M., Botterweck, G., Ventresque, A.: Learning software configuration spaces: A systematic literature review (2021). https://doi.org/10.1016/j.jss.2021.111044
19. Sayagh, M., Kerzazi, N., Adams, B., Petrillo, F.: Software configuration engineering in practice interviews, survey, and systematic literature review. IEEE Transactions on Software Engineering **46**(6), 646–673 (2018). https://doi.org/10.1109/TSE.2018.2867847

20. Schlie, A., Knüppel, A., Seidl, C., Schaefer, I.: Incremental feature model synthesis for clone-and-own software systems in matlab/simulink. In: Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A-Volume A. pp. 1–12 (2020). https://doi.org/10.1145/3382025.3414973
21. Sharif, H., Abubakar, M., Gehani, A., Zaffar, F.: TRIMMER: Application specialization for code debloating. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 329–339 (2018). https://doi.org/10.1145/3238147.3238160
22. She, S., Lotufo, R., Berger, T., Wąsowski, A., Czarnecki, K.: Reverse engineering feature models. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 461–470 (2011). https://doi.org/10.1145/1985793.1985856
23. Siegmund, N., Grebhahn, A., Apel, S., Kästner, C.: Performance-influence models for highly configurable systems. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 284–294 (2015). https://doi.org/10.1145/2786805.2786845
24. Siegmund, N., Rosenmüller, M., Kästner, C., Giarrusso, P.G., Apel, S., Kolesnikov, S.S.: Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption. Information and Software Technology **55**(3), 491–507 (2013). https://doi.org/10.1016/j.infsof.2012.07.020
25. Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Apel, S., Saake, G.: SPL Conqueror: Toward optimization of non-functional properties in software product lines. Software Quality Journal **20**(3), 487–517 (2012). https://doi.org/10.1007/s11219-011-9152-9
26. Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Saake, G.: Measuring non-functional properties in software product line for product derivation. In: 2008 15th Asia-Pacific Software Engineering Conference. pp. 187–194. IEEE (2008). https://doi.org/10.1109/APSEC.2008.45
27. Temple, P., Galindo, J.A., Acher, M., Jézéquel, J.M.: Using machine learning to infer constraints for product lines. In: Proceedings of the 20th International Systems and Software Product Line Conference. pp. 209–218 (2016). https://doi.org/10.1145/2934466.2934472
28. Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., Talwadker, R.: Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 307–319 (2015). https://doi.org/10.1145/2786805.2786852
29. Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S.: An empirical study on configuration errors in commercial and open source systems. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. pp. 159–172 (2011). https://doi.org/10.1145/2043556.2043572
30. Zhou, S., Al-Kofahi, J., Nguyen, T.N., Kästner, C., Nadi, S.: Extracting configuration knowledge from build files with symbolic analysis. In: 2015 IEEE/ACM 3rd International Workshop on Release Engineering. pp. 20–23. IEEE (2015). https://doi.org/10.1109/RELENG.2015.15
31. Ziadi, T., Frias, L., da Silva, M.A.A., Ziane, M.: Feature identification from the source code of product variants. In: 2012 16th European Conference on Software Maintenance and Reengineering. pp. 417–422. IEEE (2012). https://doi.org/10.1109/CSMR.2012.52