2017 IEEE 41st Annual Computer Software and Applications Conference

Comprehensive Static Analysis for Configurable Software via Combinatorial Instantiation

Dong Yan[1,3], Linjie Pan[2,3], Rongjie Yan[†2], Jun Yan[†2,3] and Jian Zhang[2,3]
[1]Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences
[2]State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
[3]University of Chinese Academy of Sciences
yandong14@otcaix.iscas.ac.cn, {panlj, yrj, yanjun, zj}@ios.ac.cn

***Abstract*—Equipped with customized parameters, configurable software is more flexible when facing various hardware platforms and scenario options. The configurability can tailor the source code to different instances. Consequently, it is difficult for developers to enumerate all possible configurations for finding bugs, especially for large-scale configurable software systems. In this paper, we propose a method to efficiently detect bugs of such systems with static analysis techniques. The method takes advantage of combinatorial testing techniques to generate sufficient configurations. It first extracts required parameters and the corresponding constraints from a configure file. The parameters together with constraints are employed to generate configurations with required coverage. Considering the features of configuration options, we further classify the parameters into clusters, according to the tightness of their relations. Inspired from the idea of divide-and-conquer, every cluster can be assigned with a local strength, such that the tightly coupled options can be covered, without incurring other unnecessary options. Such improvement can reduce the number of required configurations, thus improving the efficiency of static analysis. The experimental results over four real-world configurable systems demonstrate the efficiency, scalability and practicality of our method.**

***Keywords*—Combinatorial Instantiation, KConfig, Configurable Software, Graph Classification, Static Analysis.**

I. Introduction

To adapt to different hardware architectures and customize different functionalities, macro definitions are widely used in source codes of software systems, to realize compilation time configuration. The corresponding software is called configurable software, and the macro definitions are regarded as configuration options. Some descriptive languages, such as KConfig [1] and CDL [2], are developed to ease the complicated configuration. These auxiliary languages allow developers or users to modify the behavior of a software through changing configuration files rather than source files. Furthermore, they provide a constraint-based mechanism to describe the complex relations among configurable options.

Technically, different valuations of macro variables (multiple configurations) can affect the behavior of source codes and thus, may trigger various bugs. Static analysis is one of the techniques proposed for bug detection, whose input is the preprocessed code after resolving configuration options by conditional compilation. Meanwhile, it considers only a single configuration at a time. Consequently, giving multiple configurations, the performance of static analysis will be decreased. On the other hand, Reinhard Tartler et al. [3] implies that less than 80% code lines of Linux 2.6.36 are statically analyzed under the default configuration. Therefore, to perform a comprehensive analysis on a configurable software, we should maximize the expansion of configurations, which leads to an even heavier task for static analysis.

Configuration space exploration techniques [4], [5], [6], [7], [8] are essential for testing or static analysis tools to find bugs and improve configurable system quality. For example, a method to generate a configuration to maximize the coverage of code blocks in the system is proposed in [6], whose solution is calculated with SAT solvers. Sample algorithms with various coverage considerations of configuration options can also be applied for bug detection [4], [5]. Some of existing techniques are constrained to detect specific bugs, or make assumptions that might not be realistic in practice, which may not be applicable to real-world systems [5].

Given a configurable software, generating one configuration may ignore other problematic configurations during static analysis. However, generating all allowed configurations is less feasible for practical usage. A trade-off is to pick some representative configurations for analysis. According to our experience, bugs may be caused by interactions of some configuration options. To reveal such bugs, the representative configurations should cover these interactions, i.e., enumerating all the possibilities within the combinations of the configuration options. Combinatorial testing (CT) [9], [10], [11] is a mature technique that can generate a small set of test cases to cover all the feasible combinations of different parameters of a system under test (SUT). Therefore, we can apply the techniques of combinatorial coverage for configuration generation.

In this paper, to demonstrate our method for improving the quality of configurable software systems with static analysis techniques, we choose the configuration options in the format of KConfig as a typical case. We first extract the variables and their constraints in KConfig, and construct a model for configuration generation. A large-scale KConfig specification will result in a big model, which will create a large number of configurations. Therefore, the parameters in the model are

Supported in part by National Natural Science Foundation of China (Grant No. 61672505 and No. U1435220), the National Key Basic Research (973) Program of China (Grant No. 2014CB340701), and the CAS/SAFEA International Partnership Program for Creative Research Teams.

† Corresponding authors.

0730-3157/17 $31.00 © 2017 IEEE
DOI 10.1109/COMPSAC.2017.91

classified into smaller clusters, each of which can be assigned with a strength, to ensure the coverage of the corresponding parameters, with a smaller number of configurations. Finally, the global configurations together with the source code can be analyzed statically.

This paper has three main contributions. First, we apply combinatorial testing techniques in test case generation for configuration generation with certain coverage requirement. This is achieved by extracting the variables and their constraints from a configuration file and translating them into the required format for configuration generation. Second, we classify parameters into clusters with graph classification techniques, and assign local strengths to different clusters, where the scope of each strength is within the parameters of a cluster. With this improvement, we can not only explore the tightly coupled parameters more thoroughly, but also obtain a reduction on the generated configurations. Third, by integrating constraints exaction, translation, configuration generation, and static analysis process, we establish an automatic and general framework for configurable software bug detection.

The rest of the paper is organized as follows. Section 2 recaps the syntax of KConfig and the CT technique. Second 3 proposes a method to translate KConfig specification for model construction and configuration generation. Section 4 presents the details of experiments on real-world benchmarks. Section 5 comes with related work. Finally, we conclude.

II. Background

In this section, we first show a motivating example. Then we will give a brief explanation of KConfig language on its grammar and constraints. Finally, we will recap the techniques of combinatorial coverage.

A. Motivating Example

The code in Fig. 1 is a simplified fragment of an open source software system axTLS [12]. It contains two configurable parameters that start with "CONFIG" prefix. The parameters can be employed in two cases: as decision variables, or as constants. For example, the configurable parameter CONFIG_B1 is used for the compiler to choose the code block to be compiled. And CONFIG_V1 is used as a constant. These parameters increase the complexity of the source code and make it hard for manual review.

Let CONFIG_V1 be an empty string, and CONFIG_B1 be defined. Then an execution path P of the code in Fig. 1 can be extracted as:

$$P : 6 \to 8 \to 9 \to 12 \to 13 \to 18$$

Following path P, the program will refer to the pointer `ssl_ctx -> rsa_ctx`, which should be initialized in line 14-15. Consequently, such scenario leads to a NULL pointer dereference fault. However, this kind of faults can only be revealed under specific combinations of configuration options. The fact that a static analyzer can only analyze code lines enabled by macros requires one expand the macros before analyzing. So different assignments of parameters would make different code lines be analyzed. To fully detect the bugs, we should enumerate the combinations of the valuations of parameters, and then statically analyze the configurable software under various feasible configurations.

```
 1  struct SSL_CTX{
 2    uint32_t options; RSA_CTX *rsa_ctx;
 3    ...
 4  };
 5  void f1(uint32_t options){
 6    SSL_CTX *ssl_ctx = (SSL_CTX *)calloc(1,
 7                            sizeof (SSL_CTX));
 8    ssl_ctx->options = options;
 9    f2(ssl_ctx);
10  }
11  int f2(SSL_CTX *ssl_ctx){
12    uint32_t options = ssl_ctx->options;
13    if (strlen(CONFIG_V1) > 0){
14      ...
15      /*initialize the elements of ssl_ctx*/
16    }
17  #ifdef CONFIG_B1
18    uint8_t *buf = (uint8_t *)alloca(ssl_ctx->
19          rsa_ctx->num_octets*2 + 512);
20    ...
21  #endif
22  }
```

Fig. 1. A motivating example

B. KConfig Language

KConfig is a common language which is used to manage the configuration of parameters in programs. During the compilation, a configuration file will be first generated according to the KConfig file, and then passed to the `make` command as environment configuration of `Makefile`.

The KConfig language has two main responsibilities. The first is to organize the configurable options in a hierarchy structure. The second is to ensure the feasibility of the valuations of configuration options with constraints. We can easily set the value of configurable variables through targets of command `make`. Among them, the most widely used include `make menuconfig` for user-customized configuration, `make defconfig` for default configuration, and `make allyesconfig` for configuration whose configurable variables are all enabled. Detailed information can be found in documents [1], [13].

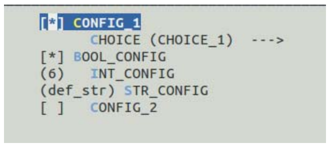
KConfig introduces variables (configuration options) to represent each parameter used as macros or constant in a C program. It also uses keywords to organize these variables as well as their attributes. The variables defined in a KConfig file can take effect in source codes and `Makefile`. In source codes, they can be used in the C preprocessor directives such as `#ifdef and #if`, and they can also be constants in the source codes which will be used in the runtime. In a `Makefile`, the configurations indicate which components will be compiled into the binaries.

The variables and their constraints depicted in a KConfig file is a KConfig model M_K. We use the example shown in Fig. 2, together with its UI in Fig. 3, to explain the features of KConfig models.

Variables A variable is defined by keyword `config` followed by its name and type. Currently KConfig language supports five types including `bool`, `tristate`, `hex`, `string`

```
menu "MENU"
  config CONFIG_1
    bool "CONFIG_1"
    default y
  choice
    prompt "CHOICE"
    depends on CONFIG_1
    default CHOICE_1
    config CHOICE_1
      bool "CHOICE_1"
    config CHOICE_2
      bool "CHOICE_2"
    config CHOICE_3
      bool "CHOICE_3"
  endchoice
  config ANON_CONFIG
    def_bool n
  config BOOL_CONFIG
    bool "BOOL_CONFIG"
    default y
    select ANON_CONFIG if CHOICE_1
  config INT_CONFIG
    int "INT_CONFIG"
    default 6 if !CONFIG_1
    default 10 if CONFIG_1
    range 5 32
    depends on BOOL_CONFIG
  config STR_CONFIG
    string
    prompt "STR_CONFIG"
    default "def_str"
    depends on BOOL_CONFIG
  if BOOL_CONFIG
    config CONFIG_2
      bool "CONFIG_2"
  endif
endmenu
```

Fig. 2. An example of KConfig



```
[*] CONFIG_1
        CHOICE (CHOICE_1)  --->
[*] BOOL_CONFIG
(6)     INT_CONFIG
(def_str) STR_CONFIG
[ ]     CONFIG_2
```

Fig. 3. The KConfig example showed in UI

and `int`, where the domain of type `tristate` contains three values, `y`, `n` or `m`. They indicate that the corresponding part will be compiled into the final binary, will not be compiled or will be compiled as module, respectively. In particular, a variable with `int` type has an optional `range` to restrict its domain. We can add an optional string to describe a variable after either the type (e.g. the string "INT_CONFIG" after type `int`) or the keyword `prompt` (e.g. the string "STR_CONFIG" after keyword `prompt`). Only those variables with type or `prompt` descriptions are visible in GUI (we call them *visible*) and thus can be modified by users. The values of other variables are not visible in the GUI (we call them *invisible*) and can be inferred from the values of visible variables, e.g. variable `ANON_CONFIG`. A variable may have one or more mutually exclusive `default` values depending on its optional conditions. Constraints may exist among variables.

Organization of variables KConfig can organize variables in a hierarchy structure that is convenient for maintenance and configuration. A menu structure, such as the menu item shown in Fig. 3, is a block of KConfig expressions between keywords `menu` and `endmenu`, and can be nested with other structures. A structure between keywords `if` and `endif` forms an *if-block*, and the keyword `if` is followed by a conditional expression. The *if-block*s are used to decide whether the contents inside them take into effect or not according to the result of the conditional expression. A *choice* block is defined by the keywords `choice` and `endchoice`, and we can put several variables of type `bool` or `tristate` into the block. Only one of those variables can take value `y`.

Constraints in KConfig We consider two kinds of constraints among different variables, i.e., dependency constraints and reverse dependency constraints. Dependency constraints are defined by the keyword `depends on`, meaning that the validity of a variable or a block depends on the value of a boolean expression. If the value is *false*, then the variable or block will not take effect. For example, v **depends on** e means that variable v is invalid if expression e is not satisfied. Similarly, reverse dependency constraints are defined by the keyword `select` which is used to assign a valid value to a variable. For example, v **select** u if e means that u is valid when v and e are both satisfied. Here, the types of u and v must be `bool` or `tristate`. Moreover, reverse dependency constraint has a higher priority over dependency constraint. That is, u can still be valid according to a reverse dependency constraint, even if it is set invalid by dependency constraint.

C. Model for Combinatorial instance construction

Combinatorial testing (CT) is a test method used in functional testing phase to test the effect of software under different combinations of parameters. It is based on the premise that most of defects in software can only arise from the interactions between or among a few parameters [14]. A test case in CT represents a combination of specific values of each parameter. When we consider the configurations covering the interference of variables in KConfig, we can apply the technique for configuration generation.

In this technique, the parameters must be enumerable. And the covering strength, which is a user-given number for the considered parameters in a combination, is usually smaller, i.e, 2 or 3. Existing extensions of CT involve the support of various kinds of constraints such as boolean expressions to restrict the values of the parameters [11], and the application of a variable covering strength that accepts multiple covering strengths for different subsets of parameters [15].

The input of CT for generating configurations covering various parameter combinations is called *combinatorial instantiation* (CI). A CI model M_I is a combination of parameters and the corresponding constraints written in boolean logic.

III. Proposed method for the analysis of configurable software systems

In this section, we will describe our method in detail. First, we need to extract variables and constraints in a KConfig model, to translate them into a CI model. Many KConfig files of real-world packages may have hundreds or even thousands of variables, which leads to big CI models. Consequently, the number of generated configurations will be huge, even if we take a small global strength for the considered number of interacted parameters. Therefore, we group the parameters in the CI model into clusters, by applying graph classification technique on an extracted graph from the generated model. And then we can put various strengths over different clusters in the CI model. Each instance generated from the CI model is validated to be a legal configuration and combined with the source code for static analysis. Finally, we can merge the bugs reported and delete duplicated ones from static analysis under various configurations.

A. Parsing Variables

The names of parameters in a CI model can be easily extracted from a KConfig model. However, it is not easy to determine their types and domains. The type of most of the variables is `bool` or `tristate`, whose domains are small discrete sets. For other types such as `string`, their domain may be very large or even infinite, which could lead to combination explosion for configuration generation. And assigning arbitrary values to variables can lead to illegal configurations. Therefore, for these types of parameters, we will pick all values that appear in M_K (the default values) to represent the domains.

Let v be a variable in an M_K, we define function $dft(v)$ to return the set of default values of v. In particular, a variable with type `tristate` can be regarded as a `bool` because `y` (compiled into binaries) and `m` (compiled as a module) have the same meaning for static analysis. The other types except for `bool` and `tristate` are translated into the corresponding types whose domains are the sets of default values in the CI model. Moreover, we add an extra value `UN` (unknown) to represent that the variable has been disabled by constraints in the configuration. For example, for the variable `INT_CONFIG` in Fig. 2, its domain in the CI model has three elements, 6, 10 and `UN`. And `INT_CONFIG` will be disabled when the constraint `BOOL_CONFIG` is unsatisfiable. We also add an empty string `EMP` to the domain of type `string`.

We summarize the rules for parsing variables of KConfig in Table I, where v is a variable in M_K, $\mathbb{T}_K$ and $\mathbb{T}_I$ denote the types in a KConfig model M_K and in the CI model M_I, respectively, and $\mathbb{D}$ represents the domain of the parameters in M_I. According to the rules, the variables of M_K shown in Fig. 2 can be translated into what Table II shows.

TABLE I
RULES FOR PARSING VARIABLES

$\mathbb{T}_K$	$\mathbb{T}_I$	$\mathbb{D}$
bool/tristate	bool	{T, F}
int/hex	int	$dft(v)$ ∪{`UN`}
string	string	$dft(v)$ ∪{`UN`, `EMP`}

TABLE II
EXAMPLE OF PARSING VARIABLES

type	name	value
string	STR_CONFIG	{"def_str", `UN`, `EMP`}
bool	CONFIG_{1,2},CHOICE_{1,2,3} ANON_CONFIG,BOOL_CONFIG	{T, F}
int	INT_CONFIG	{6, 10, `UN`}

B. Parsing Constraints

Except for dependency and reverse dependency constraints, there are also implicit constraints such as choice, variable range, and default setting in KConfig. In this paper, we focus on constraints for configuration generation.

We first introduce some notations for the ease of explanation. Let v be a variable in M_K. For every variable v, we can extract four attributes from M_K, where

- $S_d(v) = \{con | v$ depends on $con \in M_K\}$ is the set of conditions for the dependency constraints of variable v;
- $S_r(v) = \{\langle t, con\rangle | v$ select t if $con \in M_K\}$ is the set of reverse dependency constraints, representing that v selects variable t if the condition con holds;
- $S_a(v) = \{\langle val, con\rangle |$default val if $con \in M_K\} \cup \{\langle val, true\rangle |$ default $val \in M_K\}$ is the set of default values of v, meaning that the default value of v is val if condition con is true;
- $S_s(v) = \{\langle u, con\rangle | u$ select v if $con \in M_K\}$ is the set of constraints that can select v in M_K.

For example, in Fig. 2, the S_d of variable `INT_CONFIG` is {BOOL_CONFIG}, the S_r of `BOOL_CONFIG` is {⟨ANON_CONFIG, CHOICE_1⟩}, the S_a of `INT_CONFIG` is $\{\langle 6, \neg CONFIG\_1\rangle, \langle 10, CONFIG\_1\rangle\}$, and the S_s of `ANON_CONFIG` is {BOOL_CONFIG}.

We then present the translation rules for constraints of variable v according to the constraint type.

Range constraints A variable with `int` type could have an optional range condition $[low, up]$ which restricts its domain. We can rewrite the range of this constraint by:

$$v \geq low \wedge v \leq up \tag{1}$$

Default constraints For a variable to take a default value, it should be enabled. That is, either the dependency conditions hold, or some other variables can select it, denoted by $en_v = \bigvee_{con_d \in S_d} con_d \vee \bigvee_{\langle v', con_s\rangle \in S_s} v' \wedge con_s$. Then we can form the following constraint.

$$\bigwedge_{\langle val, con_a\rangle \in S_a} (en_v \wedge con_a \rightarrow (v = val)) \tag{2}$$

Dependency constraints Variable v is not disabled, only when none of variables in S_s is enabled, and the dependency constraint is not satisfied. When v is not disabled, if the type of v (denoted by T_v) is `bool` or `tristate`, we use $\neg v$ to show its disableness. Otherwise, we assign it with `UN`. Therefore, we can extract the following constraint:

$$\neg en_v \rightarrow \begin{cases} \neg v & T_v \text{ is } \texttt{bool} \text{ or } \texttt{tristate} \\ v = \texttt{UN} & \text{otherwise} \end{cases} \tag{3}$$

Reverse dependency constraints If v can select other variables, its type must be either `bool` or `tristate`. Therefore, we can safely extract

$$\bigwedge_{\langle t, con\rangle \in S_r} (v \wedge con \rightarrow t) \tag{4}$$

Choice constraints A choice block can be enabled by its dependency constraints. If a choice block is enabled, only one variable within it can be true; otherwise, all the variables appearing in the choice block will be false. We introduce an auxiliary variable *aux* with type `bool` for the availability of the choice block. The choice block is enabled **iff** all the dependency constraints of choice block are satisfied, i.e.,

$$(\bigwedge_{con \in S_d} con) \leftrightarrow aux. \tag{5}$$

Let S_c be the set of choices in a choice block. We have

$$aux = (\bigvee_{u \in S_c} u) \wedge (\bigwedge_{u \in S_c} u \rightarrow \bigwedge_{u' \in S_c, u' \neq u} \neg u') \qquad (6)$$

For example, the constraints in KConfig file described in Fig. 2 can be translated to the format shown in Table III.

TABLE III
EXAMPLE OF PARSING CONSTRAINTS

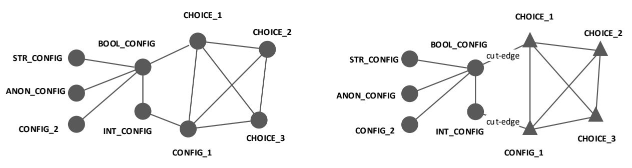
Choice	(CHOICE_1 $\vee$ CHOICE_2 $\vee$ CHOICE_3) $\wedge$ (CHOICE_1 $\rightarrow \neg$ CHOICE_2 $\wedge \neg$ CHOICE_3) $\wedge$ (CHOICE_2 $\rightarrow \neg$ CHOICE_1 $\wedge \neg$ CHOICE_3) $\wedge$ (CHOICE_3 $\rightarrow \neg$ CHOICE_1 $\wedge \neg$ CHOICE_2) = aux CONFIG_1 $\leftrightarrow$ aux
Reverse	(BOOL_CONFIG $\wedge$ CHOICE_1) $\rightarrow$ ANON_CONFIG
Dependency	$\neg$ BOOL_CONFIG $\rightarrow \neg$ ANON_CONFIG $\neg$ BOOL_CONFIG $\rightarrow \neg$ CONFIG_2 $\neg$ BOOL_CONFIG $\rightarrow$ (INT_CONFIG = UN) $\neg$ BOOL_CONFIG $\rightarrow$ (STR_CONFIG = UN)
Default	(BOOL_CONFIG $\wedge\neg$ CONFIG_1) $\rightarrow$ (INT_CONFIG = 6) (BOOL_CONFIG $\wedge$ CONFIG_1) $\rightarrow$ (INT_CONFIG = 10) BOOL_CONFIG $\rightarrow$ (STR_CONFIG ="def_str")
Range	(INT_CONFIG $\geq$ 5) $\wedge$ (INT_CONFIG $\leq$ 32)

C. Model construction with variable strengths

For real-world software systems, the scale of the generated CI model is usually large. When we assign a global strength for the model, to cover the interactions among all the parameters meeting the strength requirement, the generated configurations will be burdensome for static analysis. The fact that variables in a KConfig file are usually organized in the unit of menus makes the variables in the same menu more tightly coupled. The relation of variables from different menus is rather loose. For tightly coupled variables, we can consider various t-wise relations (variable strengths), with respect to the complexity of the code and the possibility for finding bugs. Therefore, we can group the parameters in the CI model into clusters, and assign each of them with various strengths. By using various strengths, tuples to be covered can be reduced, and the number of configurations can be further reduced.

For cluster organization, we need to build a graph to depict relationship between parameters (GRP for short) according to the CI model, where each vertex represents a parameter and two vertices are linked by an edge if the corresponding parameters appear in the same constraint. Then we can apply the graph-partition algorithm to decompose a large graph into small graphs (sub-graphs), which can be used to generate clusters. The decomposition is fulfilled by a graph data-mining software `Gephi` with a community detection method [16].

In Fig. 4(a), we present the graph of the KConfig example showed in Fig. 2. After graph construction, it can be divided into two sub-graphs, shown in Fig. 4(b), in which the vertices in the same sub-graph have the same shape. The edges connecting two sub-graphs are called cut-edges. The parameters in sub-graphs and cut-edges are organized as different clusters. That is, we can obtain four clusters after graph partition for the example, where two come from the vertices of the two sub-graphs, and the other two come from the vertices of the two cut-edges. Note that the clusters may not be mutually



(a) GRP of KConfig example (b) After classification

Fig. 4. Cluster classification

exclusive, i.e., a vertex can both belong to the clusters of a cut-edge and a sub-graph.

With the cluster information, we could assign local strengths for every cluster, in which the required coverage of the parameters can be ensured, without considering the coverage outside the cluster. We define the CI model with variable strength as follows.

Definition 1: Given a set of parameters P and their constraints C, the CI model with a set of variable strengths S is $M_S = (\mathcal{P}_c, C)$, where S is a finite set of non-negative integers, and $\mathcal{P}_c = \{(P_i, s_j) | P_i \subseteq P, s_j \in S\}$.

For a cluster P_i with strength s_j, the generated configuration of $\mathcal{P}_i$ is a tuple $((v_{i1}, val_{i1}), \ldots, (v_{ik}, val_{ik}), \ldots, (v_{in}, val_{in}))$, where $n = s_j$, $v_{ik} \in P_i$, and $val_{ik} \in D_{v_{ik}}$. That is, n-wise coverage is ensured in the cluster.

For example, considering the clusters illustrated in Fig. 4, we can generate 9 configurations from the variable strength model with strength 2 over the clusters. One of the configurations is shown in Fig. 5.

```
CONFIG_1=y
# CONFIG_2 is not set
# CONFIG_ANON_CONFIG is not set
CONFIG_STR_CONFIG="def_str"
CONFIG_INT_CONFIG=6
CONFIG_BOOL_CONFIG=y
# CONFIG_CHOICE_1 is not set
# CONFIG_CHOICE_2 is not set
CONFIG_CHOICE_3=y
```

Fig. 5. One of generated configurations

IV. EXPERIMENTS

In this section, we consider four benchmarks written in C language from open-source community to evaluate our method. We write python scripts to parse the KConfig file and translate it into a CI model. We employ CT tool ACTS [17] to process CI models. The generated configurations as well as the source codes are statically analyzed with bug detection tool Canalyze [18]. We adopt a mature tool Gephi [19] as our engine for cluster classification and visualization of sub-graphs. All the tools and scripts are run on a laptop with 8-core i7 CPU, 16GB memory and Ubuntu 12.04 64bit operating system.

In general, static analysis tools can generate a large number of bug reports while containing quite a lot of false positives. Manually confirming these bugs is a labor-intensive work. For the convenience of our experiment, we restrict Canalyze to find the bugs related to memory operations, including reference to a

NULL pointer, NULL pointers passed to non-null parameters, memory leak and memory used after free.

A. Benchmarks

We select four stand-alone software systems. All these packages are mainly written in C language and managed with KConfig files. Among the four packages, axTLS [12] is a TLSv1 SSL library designed specifically for embedded devices. And libpayload [20] is a minimal library to support stand-alone payloads that can be booted with firmware such as coreboot.

The other two packages, busybox [21] and coreboot [22], are widely used in embedded systems. There are hundreds of parameters in the corresponding KConfig files. In particular, the package coreboot supports a large number of architectures with similar functionalities. To reduce the repeated work of manually reviewing similar bug reports generated by the static analysis tool, we restrict the mainboard to `emulation` and architecture to `Qemu-X86`, which are the default values in the default configuration. With this restriction, its KConfig model only considers 799 variables and 391 constraints.

Table IV lists the numbers of variables and constraints in each benchmark. In the table, the variables are classified by their types, where #vb, #vs, #vi, #vh represent the numbers of boolean, string, integer and hex variables respectively, and #vv, #vn, #va represent the numbers of visible, invisible and all variables respectively. Similarly, constraints are also classified by their types, where #cd, #cc, #cf, #cr represent the numbers of dependency constraints, choice constraints, default constraints and reverse dependency constraints, respectively, and #ca represents the total number of constraints.

TABLE IV
VARIABLES AND CONSTRAINTS OF BENCHMARKS

Instance	SLOC	Variables			Constraints	
		#vb/#vs/#vi/#vh	#vv/#vn	#va	#cd/#cc/#cf/#cr	#ca
axTLS-1.5.3	12545	54/22/9/0	95/0	95	85/6/4/0	95
busybox-1.24.1	165254	875/23/22/0	913/7	920	149/8/545/0	702
coreboot-4.4	41077	692/33/24/50	209/590	799	264/11/0/116	391
libpayload	28512	78/0/7/9	84/10	94	64/3/11/3	81

B. The results for cluster organization

We have organized the parameters of the CI models for the four benchmarks. Their GRPs are visualized in Fig. 6, in which sub-graphs are distinguished with colors. We can observe that there are many single-vertex sub-graphs in busybox and coreboot. More precisely, after decomposition, the GRP of axTLS is divided into 14 sub-graphs, each of which has 1 to 25 variables, the GRP of busybox is divided into 237 sub-graphs, each of which has 1 to 119 variables, the GRP of coreboot is divided into 423 sub-graphs, each of which has 1 to 52 variables, and the GRP of libpayload is divided into 26 sub-graphs, each of which has 1 to 21 variables.

We present the results on cluster classification in Table V. The legends are as follows: #V, #E, #CE, #sub record the numbers of vertices, edges, cut-edges and subgraphs; and *aver*, σ, and *max* record the average, variance, and maximal number of vertices (or edges) of graphs, respectively.

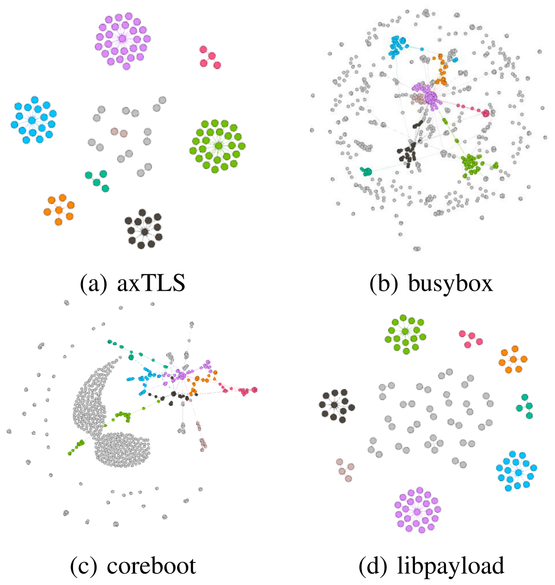

(a) axTLS (b) busybox
(c) coreboot (d) libpayload

Fig. 6. GRP decomposition

TABLE V
CLUSTER ORGANIZATION

instance	#V	#E	#sub	#CE	Vertices			Edges		
					aver	σ	max	aver	σ	max
axTLS	95	264	14	30	6.79	8.43	25	16.14	26.16	89
busybox	920	1668	237	49	3.88	10.50	119	4.87	18.85	216
coreboot	799	1232	423	78	1.89	4.36	52	3.28	17.50	173
libpayload	84	250	26	20	3.62	5.11	21	8.85	30.10	157

C. Comparison with various options

We have adopted various configuration generation options in the experimentation. The results are listed in Table VI, where *global* is for one strength over all parameters, *cluster* is from our method by applying the same strength as *global* over all clusters, *random* creates the same number of configurations as *global*, the other two are from options of `make` command. The covering strength we apply for `global` and `cluster` in the experiments is 2, based on an empirical study [5]. For other methods such as covering code lines [3], we cannot make a comparison, for the tool Undertaker fails to deal with our benchmarks. We compare the number of configurations, the covered tuples, the explored number of blocks and files, time consumption for the static analysis over all the configurations, and the detected bugs. The numbers of tuples for the last three options are not listed, for strength is not considered. Meanwhile, the numbers of blocks for coreboot and libpayload are not listed, as its configuration options are mainly used for compilation, not for block selection. Compared with applying a strength globally, the numbers of generated configurations and covered tuples are reduced greatly with cluster classification. But the numbers of explored files and blocks are similar. Therefore, we can detect bugs efficiently. For axTLS, busybox and libpayload, the number of detected bugs from the other three options are smaller. The reason being that the checked configurations are not enough to reveal potential bugs. For coreboot, we can observe that the numbers of detected bugs are the same from all the options, which are caused by the

TABLE VI
STATIC ANALYSIS RESULTS

benchmarks	elements	global	cluster	random	allyes	default
axTLS	configurations	25	19	25	1	1
	tuples	14745	2702	-	-	-
	files	63	62	61	57	26
	blocks	528	487	322	203	187
	time	1033.7	924.1	684.1	213.0	46.9
	bugs	16	16	12	12	1
busybox	configurations	44	20	44	1	1
	tuples	1656954	57958	-	-	-
	files	557	557	241	548	505
	blocks	3893	3890	810	3066	2933
	time	7028.6	2373.6	1593.9	1403.2	1720.9
	bugs	49	45	5	24	22
coreboot	configurations	95	10	95	1	1
	tuples	448939	949	-	-	-
	files	149	148	137	131	125
	blocks	-	-	-	-	-
	time	3719.6	862.7	3171.8	224.7	227.9
	bugs	15	15	15	15	15
libpayload	configurations	19	11	19	1	1
	tuples	11963	1357	-	-	-
	files	204	204	202	53	185
	blocks	-	-	-	-	-
	time	484.2	346.9	451.7	43.1	271.7
	bugs	16	16	16	2	7

included files, instead of the choice between different blocks.

`axTLS` Table VII shows 3 of 16 detected bugs and their trigger conditions. All these 3 bugs are related to the KConfig configuration, where the first bug is triggered by the value combination of the KConfig variables `SSL_PRIVATE_KEY_LOCATION` and `SSL_GENERATE_X509_CERT`.

`busybox` We present the 6 of 49 detected bugs in Table VII. The poor performance of `random` option comes from the low coverage of files and code blocks, with the randomly selected configuration options.

`coreboot` To apply static analysis over coreboot successfully, we restrict the package in two aspects. First, we forbid the compilation of the third party packages such as payload, secondary payload and IPXE. Second, we turn off the Werror option of the compiler, which is enabled in the original package. We can observe that the difference between the numbers of tuples from `global` and `cluster` is quite large. The reason being that the classification leads to many small clusters (as illustrated in Fig. 4 (c)), thus the less coverage requirements. The detected bugs are caused by NULL pointer reference and memory leaks.

`libpayload` It is an optional package in coreboot and can be compiled alone as a static library, which is not considered in our coreboot experimentation. Similar to the compiler setting of coreboot, we also disable the Werror option of compiler. Though libpayload is not compiled in coreboot, the types and the reason of the bugs are similar to those of coreboot.

D. Discussion

We present extra efforts not listed explicitly in the experimentation.

Usability of generated configurations The configurations we generate should be guaranteed legal. And the KConfig tool `conf` can help. However, a legal configuration may lead to compile or link problem (CLP). All the four benchmarks in the experimentation have CLPs under the configuration generated through command `make allyesconfig`. The configurations we generated also have these problems. For example, in the `global` option, 12 of 44 global configurations for busybox meet such problems, which are ignored.

Time consumption For each benchmark, the most time-consuming part is the multiple calls to the static analyzer. In fact, after our manual review to the different instances from same software system, we found that the difference between them are usually minor. If we can reuse the intermediate results of the previous analysis, we can effectively shorten the time of the subsequent analysis. The incremental extension can be our future work.

Bug reports We are still waiting for the official feedbacks from the package owners on these detected bugs. The published bugs only show the caused phenomena, without indicating the reason, which is difficult for us to check and confirm.

V. RELATED WORK

Complex software systems are usually configurable. The emphasis on quality improvement of such systems is increasing.

The variability between configurations and its influence on the corresponding systems are explored and certain metrics have been proposed to improve system quality [23], [24]. In the literature, researchers mainly focus on configuration space exploration techniques [6], [8], [5], where the corresponding systems can be analyzed by selecting sufficient number of configuration options. For example, the work in [6], [25] extracts the constraints from source code and the corresponding configuration file, to achieve maximized code block coverage by applying constraint solving techniques. Compared with our work, it only considers configurations with boolean valuation, ignoring other existing types. Recently, a series of sampling algorithms such as random, t-wise with already confirmed faults have also been studied, to compare their fault detection capabilities [5]. Our method is more flexible in considering local strengths for various sets of configuration options. And we apply static analysis technique over the configured source for bug detection, instead of empirical study over fixed bugs. There are some other strategies to deal with configuration-related faults, such as using combinatorial interaction testing to check different combinations of configuration options and prioritize test cases [8]. Though we also employ combinatorial testing techniques for configuration generation, the number of generated configurations could be smaller for the introduction of local strengths over different clusters. The work of [26] developed a variability-aware parser to analyze all possible configurations of Java and GNU C programs without the need of preprocessing. It mainly performs type and syntax checking, which is a subset of topics in static analysis. We employ a heavy-weight static analyzer that performs a complex checking and can find more types of hidden bugs such as memory-related ones.

TABLE VII
TRIGGER CONDITIONS OF SOME BUGS

instance	bug	description	condition
axTLS	1	dereference of NULL pointer	`SSL_PRIVATE_KEY_LOCATION=EMP` `SSL_GENERATE_X509_CERT=y`
	2		`SSL_SKELETON_MODE=y`
	3	NULL pointer passed as an argument to a nonull parameter	`HTTP_HAS_CGI=y`
busybox	1	memory leak	`ENABLE_FEATURE_DD_IBS_OBS=y` `ENABLE_FEATURE_DD_SIGNAL_HANDLING=y` `ENABLE_FEATURE_DD_THIRD_STATUS_LINE=y`
	2	NULL pointer passed as an argument to a nonnull parameter	`ENABLE_FEATURE_PREFER_APPLETS=n`
	3		`MODPROBE=y`
	4, 5, 6	dereference of NULL pointer	`FEATURE_2_4_MODULES=y`

VI. CONCLUSION

In this paper, we have proposed a general framework for bug detection of configurable systems. It first generates a set of concrete configurations from the KConfig file, and then feeds the customized system to static analyzers for detecting bugs. The work takes advantage of combinatorial testing for configuration generation with required coverage, where necessary parameters and constraints are transformed and regarded as the input of such tools. To further reduce the efforts of static analysis, the parameters extracted from a configuration file are classified into clusters, thus variable strengths can be applied for different clusters. It is worth mentioning that the extracted parameters and constraints are not limited to specific problems, as well as the combinatorial testing tools or static analysis tools. Therefore, the technique is more general and practical. In the future, we will further improve the efficiency of our method by CI model customization and incremental static analysis.

ACKNOWLEDGEMENTS

The authors would like to thank Mr. Xingming Wu, Ms. Feifei Ma and the anonymous reviewers for their helpful comments.

REFERENCES

[1] "Kconfig language," https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt.

[2] T. Berger and S. She, *Formal semantics of the CDL language*.

[3] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration coverage in the analysis of large-scale system software," *Operating Systems Review*, vol. 45, no. 3, pp. 10–14, 2011.

[4] I. Abal, C. Brabrand, and A. Wasowski, "42 variability bugs in the linux kernel: a qualitative analysis," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 421–432.

[5] F. Medeiros, C. Kästner, M. Ribeiro, R. Gheyi, and S. Apel, "A comparison of 10 sampling algorithms for configurable systems," in *Proceedings of the 38th International Conference on Software Engineering,*, 2016, pp. 643–654.

[6] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software: The 90, 000 #ifdefs issue," in *2014 USENIX Annual Technical Conference*, 2014, pp. 421–432.

[7] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki, "Mining configuration constraints: static analyses and empirical results," in *36th International Conference on Software Engineering*, 2014, pp. 140–151.

[8] C. Yilmaz, M. B. Cohen, and A. A. Porter, "Covering arrays for efficient fault characterization in complex configuration spaces," *IEEE Transactions on Software Engineering*, vol. 32, no. 1, pp. 20–34, 2006.

[9] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "An improved meta-heuristic search for constrained interaction testing," in *International Symposium on Search Based Software Engineering*, 2009, pp. 13–22.

[10] J. Lin, C. Luo, S. Cai, K. Su, D. Hao, and L. Zhang, "TCA: an efficient two-mode meta-heuristic algorithm for combinatorial test generation (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 494–505.

[11] Z. Zhang, J. Yan, Y. Zhao, and J. Zhang, "Generating combinatorial test suite using combinatorial optimization," *Journal of Systems and Software*, vol. 98, pp. 191–207, 2014.

[12] "axTLS Embedded SSL," http://axtls.sourceforge.net/.

[13] S. She and T. Berger, *Formal Semantics of the Kconfig Language*, 2010.

[14] D. R. Kuhn and M. J. Reilly, "An investigation of the applicability of design of experiments to software testing," in *Software Engineering Workshop*, 2002, pp. 91–95.

[15] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, C. J. Colbourn, and J. S. Collofello, "Variable strength interaction testing of components," in *27th International Computer Software and Applications Conference*, 2003, p. 413.

[16] V. D. Blondel, J. L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of Statistical Mechanics Theory Experiment*, vol. 2008, no. 10, pp. 155–168, 2008.

[17] L. Yu, Y. Lei, R. Kacker, and D. R. Kuhn, "ACTS: A combinatorial test generation tool," in *Sixth IEEE International Conference on Software Testing, Verification and Validation*, 2013, pp. 370–375.

[18] Z. Xu, J. Zhang, Z. Xu, and J. Wang, "Canalyze: a static bug-finding tool for C programs," in *International Symposium on Software Testing and Analysis*, 2014, pp. 425–428.

[19] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," in *Proceedings of the Third International Conference on Weblogs and Social Media*, 2009.

[20] "Libpayload," https://www.coreboot.org/Libpayload/.

[21] "Busybox," https://busybox.net/.

[22] "Coreboot," https://www.coreboot.org/.

[23] G. Ferreira, M. M. Malik, C. Kästner, J. Pfeffer, and S. Apel, "Do #ifdefs influence the occurrence of vulnerabilities? an empirical study of the linux kernel," in *Proceedings of the 20th International Systems and Software Product Line Conference*, 2016, pp. 65–73.

[24] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel, "Preprocessor-based variability in open-source and industrial software systems: An empirical study," *Empirical Software Engineering*, vol. 21, no. 2, pp. 449–482, 2016.

[25] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature consistency in compile-time-configurable system software: facing the linux 10, 000 feature problem," in *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems*, 2011, pp. 47–60.

[26] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *ACM SIGPLAN Notices*, vol. 46, no. 10. ACM, 2011, pp. 805–824.