Using Symbolic Execution to Analyze Linux KBuild Makefiles

ThanhVu Nguyen and KimHao Nguyen
University of Nebraska-Lincoln

***Abstract*—The Linux kernel can be customized and built from over 13,000 configuration options, making it one of the most highly-configurable systems in modern computing. To understand how the kernel is built over these configurations, we present a symbolic execution approach to analyze the Linux kbuild system. By exploiting several unique features in kbuild, we believe our approach can provide accurate results and scale well to the high level of configuration in Linux. Preliminary results show that our prototype can handle hundreds of build files in Linux and provide interesting information about the Linux build system.**

***Index Terms*—Linux build Systems, kbuild Makefiles, configurable systems, symbolic execution**

I. Introduction

The Linux kernel [1] is an active and large open-sourced software used in a wide range of systems, e.g., from tiny IoT sensors, mobile devices, to desktop and super computers. This flexibility is due to the configurable design of Linux, allowing the users to customize the kernel build with an extensive set of options. In total, Linux has more than 13,000 options and allows for an astronomically large number of 2^{13000} configurations[1] [2] (as a comparison, the highly-configurable Apache webserver and Firefox browser have about 1100 and 1600 options, respectively).

a) The problem: Similar to many large projects, Linux uses the *make* system [3] to control how different configuration options affect the compilation and linking of individual files to build the kernel. The make system consists of the `make` tool that takes as inputs *Makefiles*, written in the Make language and consisting of rules to build target programs. Linux has around 500 *kbuild files*, which are Makefiles with specific structures and conventions adopted by Linux developers [4]. In addition to Linux, kbuild Makefiles are used to build many other open-source, system-level software.

Kbuild files contain valuable build information for both Linux developers and users. The Linux Foundation has proposed a Google Summer of Code (GSoC) project in 2018 [5] to analyze these files to discover *build conditions* over configuration options, which indicate when source files are included in the kernel build. For example, building the kernel using a configuration satisfying the build condition α is guaranteed to use the file `file.c`.

As shown in the GSoC project, build conditions can help developers find orphan files that are never built in any configuration, test configurations that affect interested files, and determine what patches or code changes affect a given configuration. They also allow users (e.g., phone manufacturers) to estimate the kernel's build time and size, which is especially important for small devices (e.g., the "kernel tinification" project [6] is an effort to build small kernels to fit very small IoT device sensors). More generally, these conditions can reveal interesting configuration properties, e.g., the ratio of build conditions with respect to the configuration space, the complexity and types of the build conditions, "influential" options affecting how files are built, etc. These properties have been studied in various highly-configurable systems [7], [8], but not at the configuration level of Linux.

b) Prior Works: Makefiles can be considered as "programs" in the `make` declarative language and thus can be analyzed as other programs. However, compared to imperative languages such as C, there are few works on analyzing Makefiles, in particular for kbuild files. The works from [9]–[11] use patterns to analyze a subset of kbuild files, but suffer from correctness and scalability issues, e.g., fail to handle thousands of (.c) files in Linux as shown in [2]. The Kmax static analysis tool [2] improves scalability by sacrificing precision, e.g., does not support options with integer or string values (which are used often in kbuild files). This work also does not validate the resulting build conditions and analyze their uses (which are the main motivations for finding these conditions). SYMake [12] applies symbolic evaluation to general Makefiles and constructs dependency graphs for refactoring code and detecting code smell and consistencies in Makefiles. This work does not analyze build conditions and thus does not solve the build problems in the GSoC project. Moreover, this work might not scale to Linux Makefiles with many configuration options. Indeed, the GSoC project calls the analysis of Linux kbuild files a "Herculean" task due to Linux's unusually large configuration space.

c) Our approach: We propose to extend the popular symbolic execution technique [13], [14] to efficiently analyze Linux build conditions. Symbolic execution simulates program execution over symbolic inputs, forks into multiple executions corresponding to feasible conditional branches, and collects "path conditions" that are logical constraints over symbolic inputs leading to these branches. Analogously, we treat configuration options as symbolic inputs and simulate the "execution" of `make` to collect build conditions over these inputs mapping to source files.

While symbolic execution is relatively straightforward,

[1]This is a conservative estimate as many configuration options in Linux can take more than two values. Moreover, while some options or combinations are not compatible, we do not actually know what they are, and thus also need to consider those possibilites.

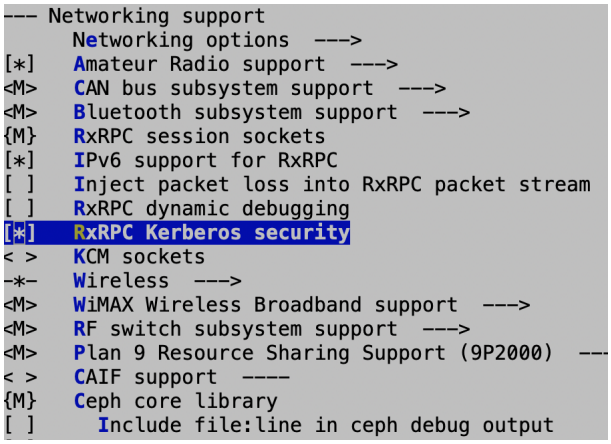

Fig. 1. Menuconfig. The notations *, M, and empty (< > or [ ]) indicate file(s) being built directly in the kernel, as a module, or not set, respectively.

many challenges arise when it is applied to real-world applications, e.g., scalability, dealing with challenging code semantics, etc. However, we can exploit several unique properties of kbuild Makefiles to make symbolic execution efficient, e.g., merging conditions mapping to similar files as the number of files is much smaller than the number of configurations.

After obtaining build conditions, we need to analyze them to solve build problems and learn interesting configuration properties. By representing symbolic states as formulae in first-order logic, we can use modern constraint solvers to solve problems such as finding orphan files and the impact of configuration options to files included in a build. For example, to find orphan files, we ask the solver for files that do not satisfy any build conditions. We can also discover other interesting configuration information about the Linux build system, e.g., the complexity of build conditions, highly-influential configuration options, etc.

d) Results: We have developed a symbolic execution tool for analyzing kbuild Makefiles. The work is in progress, but we were able to apply the tool on Busybox, another well-known system that uses kbuild Makefiles, and on many large kbuild files in Linux. We also have developed analyses to learn interesting build properties from build conditions.

II. KBuild Makefiles and Build Conditions

At a high level, the Linux kernel is built using two steps: configuring and building. In the first step, the user configures the kernel using the `menuconfig` tool shown in Figure 1. This interactive tool reads in *kconfig* files, which describe configuration options in the kernel and their dependencies (e.g., networking core driver is required for any networking device), and shows this information in a menu that the user can select from. After done configuring, the user saves the selected options and their values in a `.config` file. In the second step, the user invokes `make` to compile C source files to build the kernel image. The `make` tool, which reads in the `.config` file and kbuild files, uses `gcc` to compile and link C files based on the chosen options in `.config`.

```
obj-y := fork.o
ifeq ($A,y)
  BITS := 32
else
  BITS := 64
endif

obj-$B += probe_$(BITS).o
obj-$C += fileC.o
```

Build Conds	Files	Others
True	obj-y := fork.o	
$A = y \wedge B = y$	obj-y := probe_32.o	BITS = 32
$A = y \wedge C = y$	obj-y := fileC.o	BITS = 32
$A = y \wedge B = m$	obj-m := probe_32.o	BITS = 32
$A \neq y \wedge B = y$	obj-y := probe_64.o	BITS = 64
	...	

Fig. 2. KBuild Example.

Our work focuses on the second build step using kbuild Makefiles as there are many existing works on analyzing kconfig files (see Section V) and Linux C source code (e.g., how files are linked through preprocessing macros such as `#ifdef .. #include` [15]).

a) KBuild Makefiles: Figure 2 shows a small example of a kbuild file, adapted from [2]. The example involves three *tristate* configuration options `A,B` and `C`, which in Linux means they can take three possible values: `y` (yes), `m` (module), or unset (to be excluded from the build process). The two special variables `obj-y` and `obj-m` store the name of files being built directly to the kernel (their functionalities are always available) and files being built as modules (they can be loaded on demand), respectively.

In the example, the first assignment statement sets `obj-y` to `fork.o`. The following conditional block tests if the value of `A` equals `y`. Variables are referenced (expanded in Makefile terminology) using the $ operator. Depending on the value of A, `BITS` is set to either 32 or 64. Next is an append statement (+=). Both the left and right-hand side values of the statement are computed by first expanding variable references. The right-hand side becomes either `probe_32.o` or `probe_64.o` depending on the value of BITS, which indirectly depends on `A`. Similarly, the left-hand side value depends on `B` and becomes either `obj-y`, `obj-m`, or `obj-` (which is ignored because `B` is unset). Note that these expansions expose Makefile's unusual support for runtime variable name construction. Finally, the last assignment appends `fileC.o` to either `obj-y` or `obj-m` depending on the value of `C`.

When executed, `make` determines the source files to be included based on the values of `A,B,C`, e.g., the configuration `A=y,B=y,C=y` results in `obj-y` containing `fork.o`, `probe_32.o`, and `fileC.o`. By kbuild convention, Linux generates the object .o files by compiling the corresponding .c files (e.g., `fork.c` compiles to `fork.o`).

b) Build Conditions: To understand the relationships between configuration options and source files, we capture the semantics of the kbuild Makefiles as mappings from *build conditions*, which are constraints over configuration options, to source files (and some additional bookkeeping details). Figure 2 shows several build conditions for our running example. These mappings indicate that a kernel built with a given configuration will use source files from build conditions satisfied by that configuration. For example, the configuration `A=y,B=y,C=y` results in `obj-y` = {fork.o, probe_32.o,

fileC.o }, i.e., these files will be built directly to the kernel.

We can obtain this exact information by enumerating all possible configurations and run `make` on each of them to collect built files. However, complex build projects such as Linux would have too many configurations for this brute-force approach. Build conditions thus serve as a compact way to represent the same information.

III. Technical Approach

To analyze Linux kbuild files, we extend the bug-finding *symbolic execution* technique [13], [14]. This technique uses symbolic values for program inputs and simulates program execution over those values (instead of actual concrete values as in normal program execution). When facing a conditional branch over symbolic values, the technique considers all possible outcomes of the condition and spawns new executions for each of those outcomes. Symbolic execution remembers how it enters an execution by recording a "path condition" representing the branch condition of the execution. The user can solve the path conditions to extract concrete input values allowing the program to reach interesting program locations (e.g., those represent undesirable program behaviors).

a) Collecting Build Conditions: We use the kbuild Makefile in Figure 2 to demonstrate symbolic execution. We create three symbolic inputs representing the configuration options `A,B,C` and "simulate" `make` execution over symbolic values to collect build conditions.

The first assignment statement results in one execution with the path condition `true` mapping to `obj-y = fork.o` because we always execute this statement. The following conditional block spawns into two new execution paths corresponding to the branch conditions $A = y$ and $A \neq y$. Combining these with the original path gives two new execution paths with the path conditions `true` $\wedge$ $A = y \mapsto$ [`obj-y = fork.o, BITS=32`] and `true` $\wedge$ $A \neq y \mapsto$ [`obj-y = fork.o, BITS=64`].

These two paths reach the appending statement `obj-$B=...`, which contains unknown references (i.e., the $ operators) that need to be expanded. For the execution with path condition $A = y$ (and BITS =32), this statement spawns into three statements, e.g., obj-y += probe_32.o (when $B = y$). Combining the original execution with these produces three paths, e.g., $A = y \wedge B = y \mapsto$ `BITS = 32, obj-y =` {`fork.o, probe_32.o`}]. Similarly, the path with $A \neq y$ also becomes three new paths. Thus, we have six paths after this appending statement.

We now reach the final statement via six paths, each of which spawns into three new paths for the three values for C. In total, we accumulate $2 \times 3 \times 3 = 18$ paths representing mappings from build conditions to `.o` files being built .

b) Optimizing Symbolic Execution: Standard symbolic execution can produce an exponential number of execution paths to the number of configuration options and thus does not scale to a large number of configuration options. To deal with this issue, we extend symbolic execution with optimizations specifically for kbuild Makefiles.

A simple yet effective technique is combining execution paths [16] mapping to the same files. For example, if we have two execution paths leading to the same files (and same values for other variables), we can "merge" them into just one path by taking the disjunction of the two path conditions. While combined path conditions are more complex, modern constraint solvers can still deal with these formulae efficiently, especially when they do not involve complex arithmetics or data structures.

```
obj-$A += 1.o
obj-$B += 2.o
obj-$C += 3.o
    ...
```

In addition, kbuild Makefiles often have a sequence of append statements as shown on the right. Standard symbolic execution would enumerate all configurations as each of them updates the `obj` variables, e.g., $A = y \wedge B = y \wedge C = y \mapsto$ [obj-y = $\{1, 2, 3\}$], ... Instead, we can significantly reduce the number of paths by only storing individual files being added to `obj`, instead of the entire contents of `obj`, e.g., $A = y \mapsto [1 \in$ obj-y$], B = y \mapsto [2 \in$ obj-y$], C = m \mapsto [3 \in$ obj-m$]$. To achieve this, we can use a "split-merge" strategy [17] that splits a set of files into individual files and merges executions leading to same files. Thus, number of generated executions paths is linear, instead of exponential, in the number of individual files.

Another idea is using light-weight dataflow analysis to detect variables that are no longer used. In Figure 2 BITS is only used once and thus we can safety discard and merge execution paths involving BITS. This live-value analysis [18] can be difficult in general, but the structure of kbuild files makes it relatively simple to do (e.g., in many kbuild files, we can just "see" that some variables are only used in certain parts and never referred to again).

In general, these strategies improve scalability by limiting the number of execution paths to the number of files being built instead of the number of configurations (e.g., Linux has over 2^{13000} possible configurations but only about $38,000$ source files). Moreover, kbuild files, which are a subset of the Make language, have specific conventions and designs that favor symbolic execution. For example, they often do not contain large loops (which would require bounded symbolic execution and result in imprecision) or complex data structures (they mostly use bool, int, and string values). Also, while kbuild files can invoke arbitrary commands, many of them can be evaluated directly in symbolic execution (e.g., native pattern matching functions already model major capabilities of `grep`).

c) Using Build Conditions: After obtaining build conditions, represented as formulas in first-order logic, we can validate and analyze them to extract insightful information and answer questions about the build process. Recent advances in constraint solving [19] allow us to efficiently reason over very large formulae over many variables, i.e., the kind of complexity we would expect when analyzing formulae over thousands of Linux configuration options.

To validate these build conditions, we compare files that were used in kernel builds to files that our inferred build conditions map to. More specifically, given a configuration (e.g., default `.config` files used to build the kernels in

popular Linux distributions), we first build a kernel and collect `.o` objects created during that actual build process as "ground truths". Then we collect objects that would be created using our generated build conditions and compare them to the ground truth files to evaluate our results.

We can use the obtained conditions to solve Linux's build questions such as those in the GSoC '18 project. For example, to find orphan files that are never built in any configuration, we ask the solver for files mapped to unsatisfiable conditions or files that are not mapped to by any conditions. To find a configuration that compiles a certain file, we combine our build conditions with kconfig constraints (see Section V) and ask the solver to generate a valid configuration mapping to that file. These tasks can help Linux developers to better understand and maintain the kernel (e.g., a developer making changes to a specific file can focus on testing kernel built with configurations mapping to that file).

Moreover, these build conditions can reveal interesting properties that are commonly analyzed in highly-configurable systems. Existing works [7], [8] using systems such as Apache httpd and Coreutils show that conditions among configuration options that lead to high code coverage often involve few options and have specific forms (e.g., mostly conjunctive formulae). They also identify *influential* configuration options that must be enabled or disabled to achieve high coverage. We aim to analyze these properties from our build conditions and compare them with results from existing works.

Note that while we focus on Linux, this work can be applied to other systems that use kBuild Makefiles, e.g., Busybox [20], Zephyr RT [21], Buildroot [22]. We also believe that these ideas on using build conditions, symbolic execution, and constraint solving can be generalized to other build systems and languages [23].

IV. Preliminary Results

We are developing a prototype tool in Python. We have extended the PyMake project [24] to parse Makefiles into AST's consisting of nodes corresponding to Make statements (e.g., conditional, appending statements, rules, etc). We have implemented a breadth-first based symbolic execution algorithm over the AST nodes to collect path constraints representing build conditions. We represent these build conditions in logical formulae using the Z3 SMT solver [25].

We have applied the tool on two systems using kbuild files: Busybox [20], a software suite containing common Unix commands for embedded devices, and Linux kernel. For Busybox, which has a fairly large configuration space of 2^{500} configurations, we were able to analyze all files to obtain build conditions within 30 mins.

The tool also worked on hundreds of Linux kbuild files, many of which have large configuration spaces (e.g., those in `drivers/hid`, `driver/fs`, `input/touchscreen`, `platform/x86` contain about 100 options each). However, currently, we were not able to handle some of the largest Makefiles (e.g., we ran out of memory for Makefiles in `sound/soc/codecs`, `drivers/mfd`, `drivers/hwmon`, which contain more than 170 configuration options each). While our tool has basic merging strategies, it currently does not have the "split-merge" one described in Section III, which is designed to simplify sequences of append statements that often appear in large files (e.g., `sound/soc/codecs/Makefile` contains nearly 600 append statements).

TABLE I
Build Condition Types

Subject	conds	simple	conj	disj	mixed
Busybox	429	**326**	6	96	1
Linux	13737	**12003**	1301	271	162

We also have developed analyses to understand build conditions. Table I shows an analysis of the build conditions found by Kmax (we convert the results from Kmax to Z3 formulae and apply our analyses on these formulae). Column **conds** reports the number of build conditions, **simple** the number of "simple" conditions that are either True (length 0) or contain single configuration options, and **conj**, **disj**, and **mixed** the number of non-simple conditions (contain at least two options) that are conjunctions, disjunctions, or mixture of conjunctions and disjunctions, respectively. Thus, we found that the number of build conditions is quite small compared to the configuration space. Moreover, many build conditions are simple and involve just a single configuration option. However, while mixed conditions appear less frequently, they do exist, especially in Linux. Another analysis (data not shown) also showed that most build conditions are small, regardless of the number of configuration options. However, we found several long build conditions, e.g., Busybox contains a disjunction involving 35 configuration options Linux has a complex mixed formula involving 74 options.

V. Related Works

Other than Kmax, the works from KBuildMiner [9], GOLEM [10], Makex [11] also generate build conditions. KBuildMiner parses kbuild Makefiles with a custom grammar and collects C files matching certain usage patterns. By just parsing and not accounting for the semantics of Makefiles, KBuildMiner is imprecise and misses thousand of C files used to build the kernel [2]. GOLEM enables one or more options at a time and runs `make` to obtain build data. This approach can only cover a small number of configurations in Linux. Makex uses a similar parsing approach as KBuildMiner, but underperforms both KBuildMiner and GOLEM (yields only about 75 percent of C files in Linux according to [10]).

SYMake [12] applies symbolic execution to general Makefiles to build dependency graphs, which are useful for finding code smells and inconsistencies and refactoring (e.g., renaming variables). MAKAO [26] uses dynamic analysis to extract a dependency graph from build data observed when running `make` on a configuration and use this graph for visualization, finding dependencies among targets and built files, and detecting code smells. The works in [27], [28] show how

to improve the `make` tool (e.g., making it more efficient in parsing Makefiles). These tools are for general Makefiles and thus do not focus on kbuild files.

Several works in software product line analyze configuration options defined in kconfig files (Section II). Sincero et al. consider these options as a feature model [29] and Dintzner et al. track changes in Linux's feature model over time [30]. Berger et al. compare kconfig and the CDL modeling language to illustrate real-world use of variability modeling [31]. She et al. build a formal hierarchy of Linux features while Dietrich et al. quantify their granularity [32]. Tartler et al. compute code coverage for a single configuration and maximized coverage with a minimal set of features [33]. The Undertaker tool [34] analyzes kconfig and source code to obtain kconfig constraints and detect anomalies in source code (e.g., dead code block in C files). The work in [11] uses Undertaker's constraints with kbuild files to detect anomalies at both the file and code block levels in source code. The work in [35] combines constraints from kconfig and kbuild files to generate valid build configurations. These works mostly focus on kconfig specifications while we analyze kbuild files.

VI. Conclusion

We present a symbolic execution approach to analyze kbuild Makefiles in Linux. To handle the large configuration space in Linux, we exploit several unique properties of kbuild files to optimize the symbolic execution algorithm. While additional developments and experiments are needed, our preliminary results are promising and reveal interesting build information in Linux and Busybox.

Currently, we are implementing simplification and dataflow analyses that would significantly improve scalability. We are also working on constraint-based techniques to extract answers to build questions (e.g., finding orphan files). Finally, we are creating a GUI tool, similar to the `menuconfig` tool shown in Figure 1, to allow the users to interact with the solver to get useful build information.

Acknowledgment

We thank the anonymous reviewers for helpful comments and NSF CCF-1948536 and ARO W911NF-19-1-0054 for funding support.

References

[1] L. Torvalds, "The Linux Kernel Archives," accessed on 2020-7-3, https://www.kernel.org.

[2] P. Gazzillo, "Kmax: Finding all configurations of kbuild makefiles statically," in *FSE*. ACM, 2017, pp. 279–290.

[3] GNU, "Gnu Make," accessed on 2020-7-3, https://www.gnu.org/software/make.

[4] Kernel.org, "Linux kernel Makefiles," accessed on 2020-7-3, https://www.kernel.org/doc/Documentation/kbuild/makefiles.txt.

[5] Google Summer of Code, "Linux in Safety-Critical Systems," accessed on 2020-7-3, https://wiki.linuxfoundation.org/gsoc/2018-gsoc-safety-critical-linux.

[6] Kernel.org, "Linux Kernel Tinification," accessed on 2020-7-3, https://tiny.wiki.kernel.org/use_cases.

[7] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter, "Using symbolic evaluation to understand behavior in configurable software systems," in *ICSE*. ACM, 2010, pp. 445–454.

[8] T. Nguyen, U. Koc, J. Cheng, J. S. Foster, and A. A. Porter, "iGen: Dynamic Interaction Inference for Configurable Software," in *FSE*. ACM, 2016, pp. 655–665.

[9] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wasowski, "Feature-to-code mapping in two large product lines." in *SPLC*, 2010, pp. 498–499.

[10] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann, "A robust approach for variability extraction from the Linux build system," in *SPLC*, 2012, pp. 21–30.

[11] S. Nadi and R. Holt, "Mining Kbuild to detect variability anomalies in Linux," in *European Conference on Software Maintenance and Reengineering*. IEEE, 2012, pp. 107–116.

[12] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *ICSE*. IEEE Press, 2012, pp. 650–660.

[13] J. C. King, "Symbolic execution and program testing," *CACM*, vol. 19, no. 7, pp. 385–394, 1976.

[14] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *CACM*, vol. 56, no. 2, pp. 82–90, 2013.

[15] P. Gazzillo and R. Grimm, "SuperC: parsing all of C by taming the preprocessor," in *PLDI*, vol. 47, no. 6. ACM, 2012, pp. 323–334.

[16] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *ICSE*, 2014, pp. 1083–1094.

[17] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *FSE*, 2015, pp. 842–853.

[18] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison wesley*, vol. 7, no. 8, p. 9, 1986.

[19] C. Barrett, L. de Moura, S. Ranise, A. Stump, and C. Tinelli, "The SMT-LIB initiative and the rise of SMT (HVC 2010 award talk)," in *International Conference on Hardware and software: Verification and Testing*. Springer-Verlag, 2010, pp. 3–3.

[20] The Busybox Team, "BusyBox: The Swiss Army Knife of Embedded Linux," accessed on 2020-7-3, https://busybox.net/.

[21] The Linux Foundation, "The Zephyr Project," accessed on 2020-7-3, https://www.zephyrproject.org.

[22] Builroot project, "Buildroot: Making Embedded Linux Easy," accessed on 2020-7-3, https://buildroot.org.

[23] Wikipedia, "Build Automation Software," accessed on 2020-7-3, https://en.wikipedia.org/wiki/List_of_build_automation_software.

[24] Mozilla, "Pymake: Make implementation in Python," accessed on 2020-7-3, https://github.com/mozilla/pymake.

[25] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TCAS*. Springer, 2008, pp. 337–340.

[26] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *International Conference on Software Maintenance*. IEEE, 2007, pp. 114–123.

[27] N. Jørgensen, "Safeness of make-based incremental recompilation," in *International Symposium of Formal Methods Europe*. Springer, 2002, pp. 126–145.

[28] P. Miller, "Recursive make considered harmful," *AUUGN Journal of AUUG Inc*, vol. 19, no. 1, pp. 14–25, 1998.

[29] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk, "Is the linux kernel a software product line," in *SPLC Workshop on Open Source Software and Product Lines*, 2007.

[30] N. Dintzner, A. Van Deursen, and M. Pinzger, "Extracting feature model changes from the linux kernel using fmdiff," in *Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 2014, p. 22.

[31] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, "Variability modeling in the real: a perspective from the operating systems domain," in *Automated software engineering*. ACM, 2010, pp. 73–82.

[32] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann, "Understanding linux feature distribution," in *Workshop on Modularity in Systems Software*. ACM, 2012, pp. 15–20.

[33] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero, "Configuration coverage in the analysis of large-scale system software," in *Workshop on Programming Languages and Operating Systems*. ACM, 2011, p. 2.

[34] 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 *Conference on Computer systems*. ACM, 2011, pp. 47–60.

[35] A. Mordahl, J. Oh, U. Koc, S. Wei, and P. Gazzillo, "An empirical study of real-world variability bugs detected by variability-oblivious tools," in *FSE*, 2019, pp. 50–61.