Patchlens: A Static Analysis Tool for Variant-Aware Change Impact Analysis in Configurable Systems

Felipe de Sant'Anna Paixão
felipe.paixao@ufba.br
Federal University of Bahia
Salvador, Bahia, Brazil

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, Indiana, USA

Paulo Anselmo da Mota Silveira Neto
paulo.motant@ufrpe.br
Federal Rural University of Pernambuco
Recife, Pernambuco, Brazil

Daniel Sadoc Menasche
sadoc@ufrj.br
Federal University of Rio de Janeiro
Rio de Janeiro, Rio de Janeiro, Brazil

Gustavo Bittencourt Figueiredo
gustavobf@ufba.br
Federal University of Bahia
Salvador, Bahia, Brazil

Eduardo Santana de Almeida
eduardo.almeida@ufba.br
Federal University of Bahia
Salvador, Bahia, Brazil

Abstract

Highly Configurable Systems (HCS) generate many compile-time variants from a single codebase, making it difficult to determine which variants are affected by a code change. This paper presents PatchLens, a static analysis tool that derives Impact Conditions from source-code patches. An Impact Condition (IC) is a Boolean formula over configuration options that identifies the variants in which the changed code is present. PatchLens analyzes unified diffs, maps modified hunks to AST-level source regions, extracts preprocessor presence conditions, and resolves file-inclusion constraints through build-system resolvers. Its architecture is easily extensible through an abstract interface for integrating new build-system analyses. PatchLens currently supports large C/C++ systems such as Linux, FFmpeg, and PHP, producing human-readable formulas without compiling variants. It supports change-impact analysis, variant-aware testing, configuration-aware CI, and the generation of Impact Conditions for real-world vulnerabilities, with security vulnerability triage as one very important application. Additionally, we provide a demonstration video for the *PatchLens* tool on Zenodo (https://doi.org/10.5281/zenodo.20388433) and YouTube (https://youtu.be/AcBXvgXn4XY).

Keywords

Patch, localization, highly configurable systems, static analysis, presence condition, impact condition

1 Introduction

Highly Configurable Systems (HCS), such as the Linux kernel, FFmpeg, and PHP, derive many compile-time variants from a single codebase [6, 17, 18, 20]. Developers control these variants through configuration options, preprocessor directives, and build-system rules [5, 8, 12, 13, 17]. This variability enables software to be tailored to different platforms and requirements, but it also complicates maintenance: a source-code change may affect only the variants in which the modified code is actually present.

This patch-specific impact problem is especially challenging because variability is distributed across multiple implementation layers. A patch may modify code guarded by `#ifdef` directives, code located in a file compiled only under specific build options, or code in a subsystem that can be disabled entirely [2, 10, 11, 14, 25]. Thus, answering whether a patch affects a concrete deployment requires reasoning about both source-level and build-level variability. Conventional patch-analysis tools can identify changed lines, but they usually do not explain which configurations include those lines. Conversely, CI systems may execute many builds without knowing which configurations are actually relevant to the patch under test.

This creates practical problems for several stakeholders. Maintainers need to know whether a patch affects all variants or only a small configuration subspace [15]. CI pipelines may waste resources testing variants that cannot include the changed code. Release engineers need to assess whether deployed configurations are affected by a change. Security teams need to know whether a vulnerability patch is relevant to their configured build. In all these cases, the underlying question is the same: *which variants include the code affected by this patch?*

This paper presents *PatchLens*, a static patch-impact analysis tool for HCS. Given a source-code patch and a repository, *PatchLens* derives an *Impact Condition* (IC): a Boolean formula over configuration options that identifies the variants in which the changed code is included. For example, if a patch modifies code compiled only when `CONFIG_KVM` is enabled and the target architecture is x86, *PatchLens* can report an Impact Condition such as:

$$\texttt{CONFIG\_KVM} \wedge (\texttt{ARCH} = \texttt{x86}).$$

This formula can then be evaluated against concrete configurations to determine whether a variant is affected.

Unlike approaches that require building or checking individual variants, such as Should-I-Bother [15], *PatchLens* operates directly on "unpreprocessed" source code and unified diffs. It maps patch hunks to AST-level source regions, extracts preprocessor presence conditions, and resolves file-level inclusion constraints encoded in the build-system. The current implementation supports C/C++ codebases and includes build-system resolvers for Linux/Kbuild, FFmpeg/configure+Make, and PHP/Autotools.

The key insight behind *PatchLens* is that patch impact in configurable systems is fundamentally a variability problem [12, 13, 18, 28]: a code change affects only the variants in which the modified

artifacts are present. Therefore, *PatchLens* derives an Impact Condition that characterizes the configuration subspace impacted by a patch, enabling developers to reason about change impact across variants rather than treating the system as a single monolithic artifact.

In summary, this paper makes the following contributions:

- We present *PatchLens* as a reusable tool for deriving Impact Conditions from patches in configurable C/C++ systems.
- We describe the tool's architecture, including its patch parser, AST mapper, presence-condition extractor, build-system resolver interface, formula simplifier, and report generator.
- We provide a concise example of use showing how *PatchLens* maps a patch to a human-readable formula.
- We summarize empirical evidence showing that *PatchLens* is fast, practical, and effective on Linux, FFmpeg, and PHP.
- We discuss applications in variant-aware CI, regression test selection, vulnerability triage, feature-risk analysis, and research on variability-aware testing.

2 Background and Related Tools

In this section, we discuss foundational concepts used by *PatchLens* and compare other related tools.

2.1 Impact Conditions

A *presence condition* is a Boolean formula over configuration options that denotes the subset of variants in which a particular code artifact—such as a function, statement, file, or module—is included [1, 27, 29]. Presence conditions provide a formal link between configuration options and code inclusion semantics, and they are therefore a natural basis for reasoning about patch impact in configurable systems.

PatchLens generalizes this idea from code artifacts to patches. Given a patch p, an Impact Condition $IC(p)$ denotes the configurations in which at least one changed code region is present:

$$IC(p) = PC(h_1) \vee PC(h_2) \vee \cdots \vee PC(h_n),$$

where each h_i is a modified hunk in p and $PC(h_i)$ is the presence condition of the source region modified by that hunk.

For example, if a patch modifies two code regions, one guarded by `CONFIG_USB` and another guarded by `CONFIG_PCI`, the resulting $IC(p)$ may be:

$$\texttt{CONFIG\_USB} \vee \texttt{CONFIG\_PCI}.$$

A variant satisfying this formula includes at least one changed region and is therefore affected by the patch. This interpretation is independent of the reason for the change: the patch may implement a feature, fix a bug, refactor code, or repair a vulnerability. When the patch fixes a vulnerability, the same formula can be interpreted as the configuration subspace that contained the vulnerable code.

2.2 Related Tools

PatchLens is related to tools for patch analysis, variability-aware parsing, and build-system analysis.

Should-I-Bother. The Should-I-Bother (SiB) solution [15] checks whether specific pre-built variants are affected by a patch by matching patch hunks against variant-specific line ranges. *PatchLens* differs because it derives a unified Boolean formula over the configuration space. Once the formula is available, variant filtering becomes expression evaluation rather than per-variant patch checking.

TypeChef and SuperC. TypeChef [13] and SuperC [8] provide variability-aware parsing for unpreprocessed C. *PatchLens* is lighter-weight and more specialized: it does not attempt to construct a complete variability-aware representation of the entire program. Instead, it extracts the presence conditions needed to explain patch impact.

KBuildMiner and Kmax. KBuildMiner [3] and Kmax [7] analyze the Linux build-system to determine file presence conditions. *PatchLens* can use similar ideas, but its goal is patch-level impact analysis across source-level and build-level variability. In addition, *PatchLens* exposes a resolver interface so that support for other build-systems can be added.

Table 1 summarizes the comparison.

3 PatchLens

PatchLens takes as input a source-code repository and a patch in unified diff format. It outputs a report containing the modified files, changed regions, source-level conditions, build-level file-inclusion constraints, and the final simplified Impact Condition. This section describes the tool from a user and implementation perspective: its main functionalities, target users, usage example, architecture, and applications.

3.1 Main Functionalities and Users

PatchLens provides the following functionalities:

- **Patch parsing:** reads unified diffs and extracts file-level modifications and hunks.
- **AST-level mapping:** maps changed hunks to source-code regions in unpreprocessed C/C++ files.
- **Presence-condition extraction:** collects surrounding preprocessor guards such as `#ifdef`, `#ifndef`, `#if`, and `#elif`.
- **Build-system resolution:** resolves file-level predicates from the build-system into configuration formulas.
- **Formula simplification:** simplifies raw Boolean formulas into compact, human-readable Impact Conditions.
- **Report generation:** emits patch-impact reports that can be inspected by developers or consumed by other tools.

The tool is intended for maintainers of highly configurable C/C++ projects, CI/CD engineers designing variant-aware pipelines, release engineers assessing configuration-specific impact, security engineers triaging vulnerability patches, researchers studying variability-aware testing and sampling, and tool builders implementing support for additional build-systems.

3.2 Example of Use

Listing 1: Example command-line invocation.

```
$ patchlens \
  -e linux \
  --explain \
```

Table 1: Comparison with related tool categories.

Tool category	Patch-focused	Builds variants	Output
SiB [15]	Yes	Yes	Yes/No
TypeChef [13]	No	No	Variability-aware AST
SuperC [8]	No	No	Variability-aware AST
Kmax [7]	No	No	File-level Presence Condition
PatchLens	Yes	No	Patch-level Impact Condition

```
    analyze-git \
    ~/path/to/linux \
    ~/path/to/patch/example.patch
```

Listing 2 shows a Linux patch that modifies `arch/x86/kvm/x86.c`. This patch is used as a concrete example to illustrate how *PatchLens* computes an Impact Condition.

A user can invoke *PatchLens* by passing the target ecosystem, the repository, and the patch file, as shown in Listing 1.

For the patch in Listing 2, *PatchLens* maps the changed hunk to the corresponding AST region, extracts source-level constraints, resolves build-system constraints, and emits the Impact Condition shown in Listing 3.

Listing 3: Simplified *PatchLens* output for the patch in Listing 2.

```
(defined(CONFIG_KVM)) & ("$(ARCH)" == "x86")
```

The interpretation is straightforward: only variants compiled with KVM support for x86 include the changed code. A CI pipeline can evaluate this formula against its build configurations and prioritize only the affected jobs. If the patch fixes a vulnerability, a security team can evaluate the same formula against deployed configurations to decide whether the deployment was exposed.

Figure 1 shows a screenshot with an example *PatchLens* command and its respective output.

3.3 Architecture

Figure 2 illustrates the pipeline implemented by *PatchLens* to derive Impact Conditions from source-code patches. The architecture is organized around five main modules: the patch parser, the AST mapper, the presence-condition extractor, the build-system resolver interface, and the formula simplifier. Together, these modules transform a unified diff into a compact Boolean formula that characterizes the configuration subspace affected by the patch. The core pipeline is independent of any specific build-system; build-system-specific knowledge is encapsulated behind the resolver interface.

Patch parser. The patch parser is responsible for reading unified diff patches and extracting their file-level modifications and hunks. Each hunk identifies a localized source-code change, including the affected file, the modified line ranges, and the added or removed code. This information provides the entry point for the rest of the pipeline, since subsequent modules operate over the source regions modified by each hunk.

AST mapper. The AST mapper parses the affected unpreprocessed C/C++ source files and maps each changed hunk to the corresponding AST-level source region. *PatchLens* uses a lightweight Tree-sitter-based parser [16] because it does not require a complete variability-aware AST, such as the ones generated by TypeChef [13]. Instead, it needs enough syntactic structure to locate the AST nodes associated with the changed hunks and to relate these nodes to the surrounding source context.

This design keeps the parser simple and robust. If *PatchLens* cannot precisely interpret a local construct, it falls back to a conservative condition. Conservative over-approximation may include extra variants, but it avoids omitting variants that may contain the changed code.

Presence-condition extractor. The presence-condition extractor traverses from each changed AST node to the translation-unit root and collects enclosing preprocessor guards, such as `#ifdef`, `#ifndef`, `#if`, and `#elif`. The collected guards are conjoined to form the source-level presence condition of the changed region. If a patch modifies multiple hunks, *PatchLens* disjoins their hunk-level conditions because a variant is affected whenever it includes at least one changed region.

Build-system resolver interface. Source-level analysis alone is insufficient because entire files may be conditionally compiled by the build-system, such as Kbuild, CMake, or Make. To represent this dependency, *PatchLens* analyzes build-system code through specific Build-system Resolver implementations.

The build-system resolver interface is responsible for resolving these symbolic file-inclusion rules into Boolean formulas over configuration options. In other words, it abstracts the question of whether a source file is compiled under a given configuration. This abstraction is necessary because, while C/C++ source code follows language standards and can therefore be parsed in a largely build-system-agnostic way, the C/C++ languages do not define a standard build-system. Consequently, build-time variability must be analyzed through specific resolvers that encode the conventions and semantics of each target build-system.

Listing 4: Conceptual resolver interface.

```
def find_required_configs(
    repository_path: str,
    target_file_path: str,
) -> Flag:
    ...
```

The resolver interface is intentionally abstract. Conceptually, a resolver implements the operation shown in Listing 4: it takes

Listing 2: Contents of a Linux patch that fixes a vulnerability inside the KVM subsystem (CVE-2023-1513).

```
1  --- a/arch/x86/kvm/x86.c
2  +++ b/arch/x86/kvm/x86.c
3  @@ -5263,12 +5263,11 @@ static void kvm_vcpu_ioctl_x86_get_debugregs(struct kvm_vcpu *vcpu,
4   {
5     unsigned long val;
6
7  + memset(dbgregs, 0, sizeof(*dbgregs));
8     memcpy(dbgregs->db, vcpu->arch.db, sizeof(vcpu->arch.db));
9     kvm_get_dr(vcpu, 6, &val);
10    dbgregs->dr6 = val;
11    dbgregs->dr7 = vcpu->arch.dr7;
12 - dbgregs->flags = 0;
13 - memset(&dbgregs->reserved, 0, sizeof(dbgregs->reserved));
14  }
```

Figure 1: Example screenshot of *PatchLens* execution and Impact Condition generation.

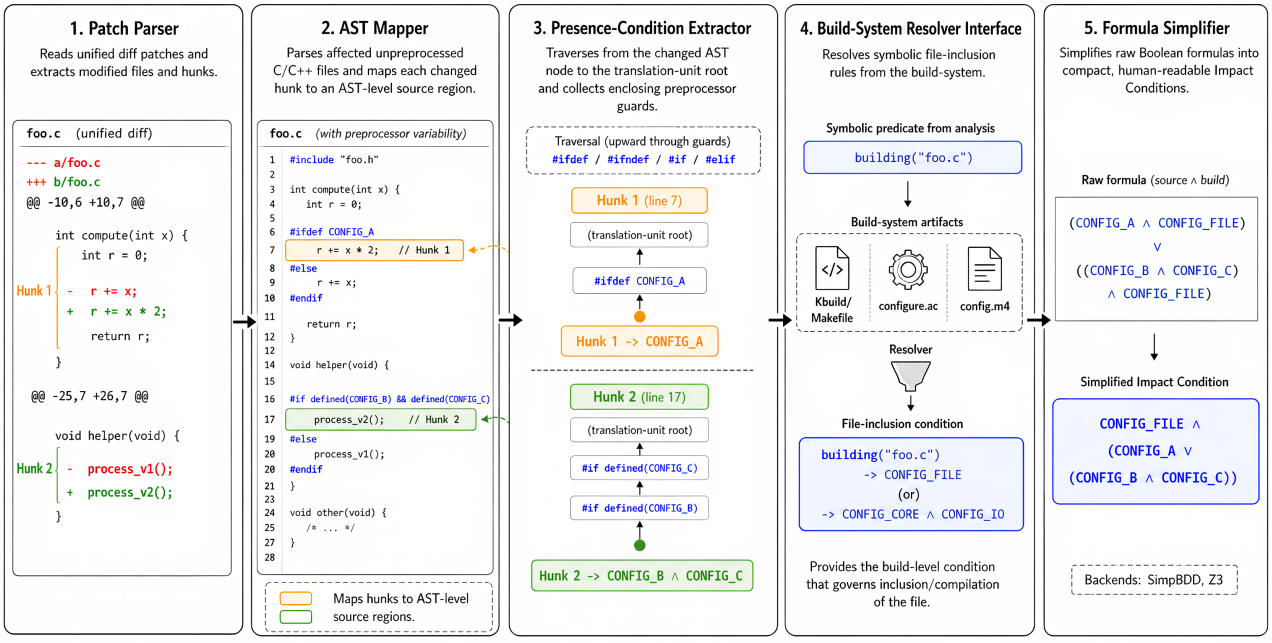

Figure 2: Architecture of the *PatchLens* tool and its respective modules.

the paths to the repository and target file as input and returns the corresponding presence condition, represented by the `Flag` type.

This design makes *PatchLens* extensible: to support a new system, developers can implement a resolver that maps source files to their build conditions. The rest of the pipeline remains unchanged.

Existing resolvers. PatchLens currently includes resolvers for three large configurable systems.

Linux/Kbuild. The Linux resolver analyzes Kbuild and Makefile rules. Given a target file, it finds the nearest build file, extracts the local predicate that includes the corresponding object, and then ascends through parent directories to collect directory-level inclusion conditions. Linux tristate options are handled by treating both built-in and module selections as inclusion, since both compile the target code.

PHP/Autotools. The PHP resolver analyzes common Autotools patterns in `config.m4`, `configure.ac`, and related files. It recognizes macros such as `PHP_NEW_EXTENSION`, `PHP_ADD_SOURCES`, `PHP_ARG_ENABLE`, and `PHP_ARG_WITH`, mapping extension declarations and configure flags to file-level conditions.

FFmpeg/configure+Make. The FFmpeg resolver analyzes configure-driven Makefile assignments, such as `OBJS-$(CONFIG_X)` and related object-list patterns. It maps source files to the feature options that include their corresponding object files and propagates component-level guards from FFmpeg's library and tool directories.

Formula simplifier. The formula simplifier receives the raw Boolean formula produced by the previous modules and rewrites it into a compact, human-readable Impact Condition. This step is important because formulas obtained from source-level guards and build-system constraints may contain redundant subexpressions [29]. *PatchLens* currently supports simplification through SimpBDD [29] and z3 [4] backends, producing formulas that are easier to inspect manually and simpler to consume in downstream workflows such as CI, testing, and vulnerability triage.

3.4 Applications

Variant-aware change triage. *PatchLens* helps maintainers distinguish between system-wide changes and configuration-specific changes. If the Impact Condition simplifies to true, the patch affects all variants. Otherwise, maintainers can inspect the formula to understand which configuration options gate the change.

Configuration-aware CI. CI systems can evaluate Impact Conditions against build configurations. If a job's configuration does not satisfy the Impact Condition, the job does not include the changed code and can be deprioritized or skipped, depending on the project's testing policy. This enables faster feedback without requiring the CI system to infer patch relevance from file paths alone [15].

Regression test selection. Regression testing aims to run tests affected by a change [26]. In configurable systems, this requires knowing not only which code changed, but also which configurations include that code. *PatchLens* provides the configuration side of this mapping. Test-selection tools can combine coverage data with Impact Conditions to identify relevant test/configuration pairs.

Variant sampling and fuzzing. Researchers and practitioners can use Impact Conditions to guide sampling and fuzzing toward relevant subspaces [9, 11, 19]. Instead of sampling uniformly across the full configuration space, a testing strategy can prioritize variants satisfying historical or current impact formulas.

Security vulnerability triage. For a patch to successfully fix a vulnerability, it must affect all vulnerable variants of the system. Therefore, by calculating the Impact Condition of a vulnerability patch, one can identify the variants affected by the vulnerability [21]. This can enrich vulnerability advisories with compile-time requirements. For example, instead of saying only that a Linux version is affected, an advisory could additionally state that exposure requires `CONFIG_KVM` and x86. This helps operators assess whether their deployed build is actually vulnerable.

4 Validation

We validate *PatchLens* by assessing whether it can efficiently derive Impact Conditions for real-world patches in large configurable systems. Our goal is not to evaluate the tool only as a vulnerability-analysis technique, but to validate its practicality as a general patch-impact analysis tool for configurable C/C++ systems.

4.1 Dataset and Experimental Setup

To perform this validation, we first constructed a large-scale dataset of real-world patches. We collected patches from two complementary sources. First, we processed the NVD 2.0 JSON feeds [1] from 2002 to July 2025 and selected references that point directly to GitHub commits. For each matching reference, we retrieved the corresponding patch in unified diff format. This process yielded 12,392 patches. Second, we incorporated the PatchDB dataset [30], which contains 4,076 patches collected from multiple sources. After removing duplicates between both sources, the final dataset contained 14,897 patches from approximately 4,000 software projects.

We then selected Linux, FFmpeg, and PHP as target systems for validation. These systems were among the ten projects with the largest number of patches in our dataset and, more importantly, they exercise different variability mechanisms and build-system designs. Linux combines preprocessor variability with Kbuild/Kconfig rules; FFmpeg uses configure-time options together with GNU Make; and PHP relies on Autotools-based build scripts. Thus, these systems provide a diverse validation setting for assessing whether *PatchLens* can derive Impact Conditions across different configurable C/C++ ecosystems.

Finally, we executed *PatchLens* on all selected patches and recorded whether the tool successfully computed an Impact Condition, the reason for failures, and the average processing time.

4.2 Results

Table 2 summarizes the validation results. The overall success rates were high: *PatchLens* successfully computed Impact Conditions for most analyzed patches in Linux, FFmpeg, and PHP. Average processing time remained below 100ms per patch in all three systems, suggesting that *PatchLens* is efficient and lightweight enough for large-scale analyses and integration into development workflows such as continuous integration.

[1] The NVD 2.0 JSON feeds are yearly machine-readable vulnerability records published by the National Vulnerability Database, available at https://nvd.nist.gov/vuln/data-feeds.

Table 2: *PatchLens* results across projects.

Project	Total	Success	Failure	Avg time (ms)
Linux	1,192	1,057 (≈ 88.67%)	135 (≈ 11.32%)	≈ 47.4ms
FFmpeg	289	280 (≈ 96.88%)	9 (≈ 3.11%)	≈ 86.3ms
PHP	100	68 (68%)	32 (32%)	≈ 87.1ms

The generated formulas were also compact. On average, the Impact Conditions contained 1.84 configuration variables for Linux, 3.23 for FFmpeg, and 1.04 for PHP. This compactness is important because *PatchLens* is intended not only to support automated workflows, but also to produce human-readable explanations of patch impact. In practice, most generated formulas can be directly inspected by maintainers and easily evaluated against concrete build configurations.

Overall, these results indicate that *PatchLens* can derive accurate and concise Impact Conditions for real-world patches in large highly configurable systems, while remaining fast enough for interactive or CI-based workflows. The validation also shows that the resolver-based architecture is practical across substantially different build-systems, supporting the use of *PatchLens* as a general tool for configuration-aware patch impact analysis.

4.3 Limitations

PatchLens currently targets C/C++ systems with compile-time variability. It does not directly model runtime variability or language ecosystems where configuration is primarily resolved at execution time.

The main source of project-specific effort is build-system support. Although the core architecture is build-system agnostic, each new build-system requires a resolver capable of mapping source files to file-inclusion conditions. Complex build constructs, code generation, shell escapes, and nonstandard macros may lead to unresolved cases or conservative over-approximation.

Another limitation is header-file impact. A change to a header may affect all translation units that include it, directly or indirectly. *PatchLens* can analyze the changed header itself, but richer transitive include analysis would improve precision for header-heavy patches.

5 Conclusion

PatchLens makes patch impact configuration-aware. Given a source-code patch in a highly configurable C/C++ system, it derives a Boolean Impact Condition that identifies the variants in which the changed code is present. This enables maintainers, CI systems, researchers, and security teams to reason about patch relevance without compiling variants.

By separating generic patch/source analysis from project-specific build-system resolvers, *PatchLens* provides an extensible architecture for analyzing large configurable systems. Its current support for Linux, FFmpeg, and PHP demonstrates that the approach is practical across different build ecosystems. Beyond vulnerability triage, *PatchLens* supports broader use cases in change-impact analysis, variant-aware testing, configuration-aware CI, feature-risk analysis, and empirical research on configurable software systems.

Future work includes support for additional build-systems such as CMake, Meson, and Bazel; deeper include-dependency analysis; IDE integration; richer visualization of formulas; and direct integration with CI and regression-test-selection tools.

Artifact Availability

PatchLens is released as open-source software under the MIT license. The *PatchLens* Artifact [24] at https://doi.org/10.5281/zenodo.22211457 includes the source code, documentation, example patches, resolver implementations, and scripts for reproducing the main analyses. In addition to the artifact, we also provide a GitHub repository [22] at https://github.com/fleap-dev/patchlens for continuous updates to the tool.

Lastly, we also provide a video demonstrating the Patchlens tool in Zenodo [23] and at https://youtu.be/AcBXvgXn4XY.

Acknowledgments

This work was partially supported by CNPq grant 444956/2024-7, and FAPESB INCITE PIE0002/2022 grant. This work was also partially supported by CAPES and FAPERJ under grants E-26/204.268/2024 and E-26/260.168/2026, CNPq grants 444956/2024-7, 424622/2021-1 and 315106/2023-9, as well as Finep PlatCiber.

References

[1] Iago Abal, Claus Brabrand, and Andrzej Wasowski. 2014. 42 variability bugs in the Linux Kernel: a qualitative analysis. In *Proc. of the 29th ACM/IEEE Intl. Conf. on Automated Software Engineering (ASE)*. 421–432. doi:10.1145/2642937.2642990

[2] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Größlinger, and Dirk Beyer. 2013. Strategies for Product-Line Verification: Case Studies and Experiments. ICSE '13. (2013).

[3] Thorsten Berger and Steven She. 2012. Google Code Project: various variability extraction and analysis tools. http://code.google.com/p/variability/ Visited on 2025-06-12.

[4] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, 337–340.

[5] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. SPLC '12. (2012). doi:10.1145/2362536.2362544

[6] Alejandra Garrido and Ralph Johnson. 2005. Analyzing Multiple Configurations of a C Program. ICSM '05. (2005).

[7] Paul Gazzillo. 2017. Kmax: finding all configurations of Kbuild makefiles statically. In *Proc. 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, New York, NY, USA, 279–290. doi:10.1145/3106237.3106283

[8] Paul Gazzillo and Robert Grimm. 2012. SuperC: Parsing All of C by Taming the Preprocessor. PLDI'12. (2012).

[9] Lukas Güthing, Mathis Weiß, Ina Schaefer, and Malte Lochau. 2024. Sampling Cardinality-Based Feature Models. In *Proc. 18th International Working Conference on Variability Modelling of Software-Intensive Systems* (Bern, Switzerland) *(VaMoS '24)*. ACM, New York, NY, USA, 46–55.

[10] Muhui Jiang, Jinan Jiang, Tao Wu, Zuchao Ma, Xiapu Luo, and Yajin Zhou. 2024. Understanding Vulnerability Inducing Commits of the Linux Kernel. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 170:1–170:31.

[11] Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. 2012. An algorithm for generating t-wise covering arrays from large feature models. In *Proc. 16th International Software Product Line Conference-Volume 1*. 46–55.

[12] Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. 2011. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. ACM, New York, NY, USA, 805–824. doi:10.1145/2048066.2048128

[13] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. 2010. TypeChef: toward type checking #ifdef variability in C. In *FOSD* (Eindhoven, The Netherlands). ACM, New York, NY, USA, 25–32.

[14] Christian Kästner, Thomas Thüm, Gunter Saake, Janet Feigenspan, Thomas Leich, Fabian Wielgorz, and Sven Apel. 2009. FeatureIDE: A Tool Framework for Feature-Oriented Software Development. SPLC '09. (2009).

[15] Tobias Landsberg, Christian Dietrich, and Daniel Lohmann. 2024. Should I Bother? Fast Patch Filtering for Statically-Configured Software Variants. SPLC '24. (2024). doi:10.1145/3646548.3672585

[16] Afshan Latif, Farooque Azam, Muhammad Waseem Anwar, and Amina Zafar. 2023. Comparison of Leading Language Parsers – ANTLR, JavaCC, SableCC, Tree-sitter, Yacc, Bison. In *2023 13th International Conference on Software Technology and Engineering (ICSTE)*. 7–13. doi:10.1109/ICSTE61649.2023.00009

[17] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An analysis of the variability in forty preprocessor-based software product lines. In *Proc. 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 105–114. doi:10.1145/1806799.1806819

[18] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable Analysis of Variable Software. ESEC/FSE '13. (2013). doi:10.1145/2491411.2491437

[19] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. 2016. A Comparison of 10 Sampling Algorithms for Configurable Systems. ICSE '16. (2016).

[20] Flávio Medeiros, Márcio Ribeiro, et al. 2013. Investigating Preprocessor-Based Syntax Errors. GPCE '13. (2013).

[21] Felipe de Sant'Anna Paixão, Joanna C. S. Santos, Paulo Anselmo da Mota Silveira Neto, Daniel Sadoc Menasche, Gustavo Bittencourt Figueiredo, and Eduardo Santana de Almeida. 2026. Automated Detection of Configuration-Specific Security Vulnerabilities via Patch Analysis. *Proc. ACM Softw. Eng.* FSE (July 2026). doi:10.1145/3808126

[22] Felipe Paixão. [n. d.]. Patchlens Repository. https://github.com/fleap-dev/patchlens. n. d..

[23] Felipe Paixão. 2026. PatchLens Tool Demo Video. doi:10.5281/zenodo.20388433

[24] Felipe Paixão. 2026. *[SBES-Tools] PatchlensTool.* doi:10.5281/zenodo.22211457

[25] Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves le Traon. 2010. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *2010 Third International Conference on Software Testing, Verification and Validation*. 459–468. doi:10.1109/ICST.2010.43

[26] G. Rothermel and M.J. Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering* 22, 8 (1996), 529–551. doi:10.1109/32.536955

[27] Reinhard Tartler et al. 2011. Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *Conf. on Computer Systems (EuroSys '11)*. ACM, 47–60. doi:10.1145/1966445.1966451

[28] Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2014. Static Analysis of Variability in System Software: The 90,000 #ifdefs Issue. USENIX ATC '14. (2014).

[29] Alexander Von Rhein, Alexander Grebhahn, Sven Apel, Norbert Siegmund, Dirk Beyer, and Thorsten Berger. 2015. Presence-Condition Simplification in Highly Configurable Systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 178–188. doi:10.1109/ICSE.2015.39

[30] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. Patchdb: A large-scale security patch dataset. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 149–160.