Should I Bother? Fast Patch Filtering for Statically-Configured Software Variants

Tobias Landsberg
landsberg@sra.uni-hannover.de
Leibniz Universität Hannover
Hannover, Germany

Christian Dietrich
dietrich@ibr.cs.tu-bs.de
Technische Universität Braunschweig
Braunschweig, Germany

Daniel Lohmann
lohmann@sra.uni-hannover.de
Leibniz Universität Hannover
Hannover, Germany

ABSTRACT

In the face of critical security vulnerabilities, patch and update management are a crucial and challenging part of the software life cycle. In software product families, patching becomes even more challenging as we have to support different variants, which are not equally affected by critical patches. While the naive "better-patched-than-sorry" approach will apply all necessary updates, it provokes avoidable costs for developers and customers.

In this paper we introduce SiB (Should I Bother?), a heuristic patch-filtering method for statically-configurable software that efficiently identifies irrelevant patches for specific variants. To solve the variability-aware patch-filtering problem, SiB compares modified line ranges from patches with those source-code ranges included in variants currently deployed. We apply our prototype for CPP-managed variability to four open-source projects (Linux, OpenSSL, SQLite, Bochs), demonstrating that SiB is both effective and efficient in reducing the number of to-be-considered patches for unaffected software variants. It correctly classifies up to 68 percent of variants as unaffected, with a recall of 100 percent, thus reducing deployments significantly, without missing any relevant patches.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Software configuration management and version control systems**.

KEYWORDS

Software Product Lines, Software Evolution, Patch Filtering

ACM Reference Format:
Tobias Landsberg, Christian Dietrich, and Daniel Lohmann. 2024. Should I Bother? Fast Patch Filtering for Statically-Configured Software Variants. In *28th ACM International Systems and Software Product Line Conference (SPLC '24), September 02–06, 2024, Dommeldange, Luxembourg*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3646548.3672585

1 INTRODUCTION

Every few years a catastrophic security vulnerability is discovered in a widely-deployed software component, posing significant risks and costs [4, 12, 15, 49]. For instance, in 2014 the Heartbleed vulnerability [26] impacted an estimated 24–55 percent of popular HTTPS sites [12], causing widespread concern and necessitating rapid patch deployment. The response was swift, with all Alexa top 500 sites applying the security patch within 48 hours. However, patch application came at a cost, as exemplified by the estimated *monthly* $400 000 cost alone for certificate revocation incurred by CloudFlare's primary CA partner [32]. Given that Heartbleed originated from a bug in an *optional* software feature that is enabled via a *C preprocessor (CPP)* configuration switch (see Lst. 1), it is worth questioning whether the costly and hurried patch application was necessary for all OpenSSL deployments.

```
--- a/ssl/d1_both.c
+++ b/ssl/d1_both.c
@@ -1459,26 +1459,36 @@
  #ifndef OPENSSL_NO_HEARTBEATS
  int tls1_process_heartbeat(SSL *s) {
  [...]
-       /* Read type and payload length first */
-       hbtype = *p++;
-       n2s(p, payload);
-       pl = p;
        if (s->msg_callback) [...]
+       /* Read type and payload length first */
+       if (1 + 2 + 16 > s->s3->rrec.length)
+               return 0; /* silently discard */
+       hbtype = *p++;
+       n2s(p, payload);
+       if (1 + 2 + payload + 16 > s->s3->rrec.length)
+               return 0; /* silently discard */
+       pl = p;
  [...]
 #endif
```

Listing 1: Patch for Heartbleed vulnerability [26]. The patch was only necessary for deployments that had the OpenSSL heartbeat extension enabled.

Like OpenSSL, most system software is configured at compile time to tailor it with respect to a broad range of supported hardware architectures and application domains. The Linux kernel, as a prime example of a highly configurable system, provides more than 17 000 configurable features [22], which are used to drastically reduce its size and attack surface by tailoring it, for example, for specific cloud VM settings [16] or embedded appliances [37]. In the embedded domain, it is common that a single vendor provides and maintains dozens or hundreds of customer-specific statically configured product variants, which are often [6, 33] managed as *software product lines (SPLs)* [8] built on top of OSS components, like Linux, OpenSSL, and SQLite. Whenever a new (critical) patch arrives for one of these components, there is a high probability that the patch targets a feature that is *not* included in your own, your customers' or at least *some* of your customers' variants [28, 29]. This raises the question: Should I bother?

SPLC '24, September 02–06, 2024, Dommeldange, Luxembourg

ACM ISBN 979-8-4007-0593-9/24/09
https://doi.org/10.1145/3646548.3672585

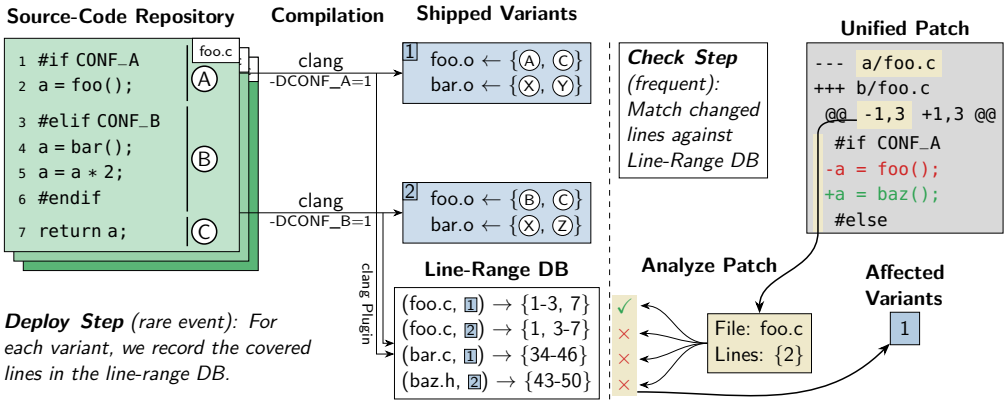


Figure 1: Overview of the SiB approach. In the *line-range database (LRDB)*, we record, with a C-preprocessor plugin, the source-code lines that are compiled for each shipped variant. When a new patch comes in, we extract the modified source lines and consult the LRDB for the affected variants.

In this paper we introduce SiB (Should I Bother?), a heuristic patch-filtering method for statically-configured software that safely identifies irrelevant patches for specific variants. We focus on annotation-based [44] variability (i.e., conditional compilations), as it is one of the most common approaches for configurable software [6]. In contrast to variability-aware functional-equivalence checking [47], we also do not require a formal feature model (often not available for most OSS components) by restricting our analysis to the relevant variant set. Furthermore, we also support C++ projects. In particular, we claim the following contributions:

- We define the variability-aware patch-filtering problem for statically-configured programs.
- We propose SiB, a fast and safe patch-filtering method based on covered line ranges for static annotation-based variability and provide a prototype for CPP-managed variability.
- We evaluate our approach with four statically-configurable open source projects (Linux, OpenSSL, SQLite, Bochs) and demonstrate that SiB is scalable, effective, and efficient by reducing deployments by up to 68 percent (Linux).

The rest of this paper is structured as follows: In Sec. 2 we describe our system model and define the variability-aware patch-filtering problem, for which we propose SiB in Sec. 3. We evaluate our approach in Sec. 4 on four open-source projects and discuss our results in Sec. 5. In Sec. 6 we discuss related work and conclude this paper in Sec. 7.

2 PROBLEM DESCRIPTION

In the following, we introduce our system model and the concept of variability-aware patch-filtering for statically-configured software.

2.1 System Model

Statically-configured software refers to projects where feature selection takes place during compile time. In this setting, a single *source repository* is used to generate multiple software *variants* by selectively enabling or disabling features during the compilation process. A set of selected features is called a *configuration*. Although this kind of static variability results in a potentially exponential number of variants, we *restrict* our interest to scenarios with a known set of variants, for example, comprising those variants that actually have been shipped to customers. Hence, we refer to these variants of interest as the *shipped variants*.

Our focus is on static *annotation-based* variability (i.e., conditional compilation): Depending on the configuration, we select a subset of all files and within those configuration-dependent files, we pass only the parts relevant to the chosen features into the compilation process to produce a configuration-specific software artifact. Annotation-based variability mechanisms for conditional compilation are available in various build systems and languages, with the *C preprocessor (CPP)* being the defacto standard.

In C/C++ the CPP establishes a static variability model based on CPP macros: For each feature, there is a CPP macro that influences the preprocessor directives (#ifdefs), which determine whether code blocks are included or excluded from the compilation process. Consequently, a set of defined CPP macros corresponds to a configuration. For instance, in Fig. 1, the macro `CONF_A` decides whether block A or block B is included in the compilation.

In addition to the CPP, other programming languages support similar variability mechanisms: For example, in Rust the `#[cfg]` function annotation controls the inclusion or exclusion of functions based on a compile-time predicate. This enables conditional compilation analogous to CPP variability but only at function granularity.

Our approach also encompasses changes to the source-code repository: The repository records all changes to the software and can produce a *patch* as a textual difference between two repository states (i.e., in the unified diff format). Therefore, patches can represent not only individual changes but also the difference between two repository states, for example, the last shipped and the current state. Hence, we cover non-linear development histories and are compatible with distributed version control systems (i.e., `git`).

In summary, our system model targets compiled languages with statically-configured, annotation-based software variability. We focus on known variant sets, rely on patches to summarize single or multiple changes, and apply our method to CPP-managed variability in C and C++ projects.

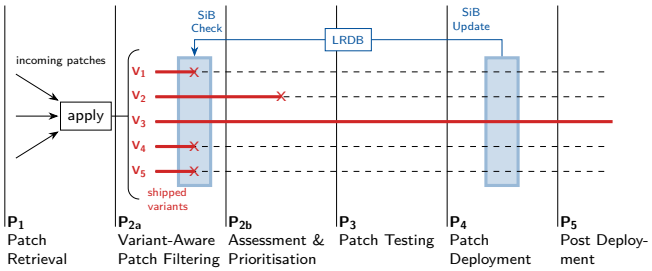


Figure 2: Variant-aware security patch management process. SiB extends P_2 by filtering variants using information gathered for the most recently shipped version in P_4.

2.2 Patch Management

In the literature, the process of analyzing and applying patches for rolled-out software is often denoted as *patch management* [11]. Patch management plays a crucial role in the software life cycle and is usually described as a five-stage process (Fig. 2):

- **P_1** *Patch Retrieval*: A patch becomes available. Whether it is from upstream or self-developed, is irrelevant to our approach.
- **P_2** *Assessment & Prioritization*: The maintainer needs to (manually) assess the patch and prioritize its deployment in an update if necessary. In a multi-variant scenario, even determining whether the code changes are used in any variant is a non-trivial task.
- **P_3** *Testing*: The maintainer has to test the update (e.g., in a staging environment). This usually involves running all tests. In a multi-variant scenario, this has to be repeated for every variant.
- **P_4** *Deployment*: The update is deployed to production systems, internal or customer-owned. This may lead to downtimes.
- **P_5** *Post Deployment*: The patched system has to be monitored for any unexpected service disruptions.

While every step in the process incurs its own costs that should be minimized whenever possible, P_4 and P_5 represent particularly high costs and risks. This is one of the reasons why patches are frequently accumulated until a new release is deployed, which, however, is often not feasible for high-priority patches (i.e., security fixes). In multi-variant scenarios, the costs and risks are further amplified by the number of variants.

2.3 Variability-Aware Patch Filtering

In the case of maintaining a configurable software with multiple shipped variants, the software life cycle fans out for each patch and variant (see Fig. 2). This multiplication of effort for the patch-management process necessitates stopping the process as early as possible for as many branches as possible to optimize resource utilization and reduce the work for tools running later in the process [34]. Therefore, SiB integrates directly after retrieving a patch by splitting the patch assessment step P_2. SiB employs automatic patch filtering, depicted as step P_{2a} in Fig. 2, which means traditional assessment P_{2b} (and following steps) must only be performed on a reduced set of configurations, if any. For this purpose SiB utilizes information gathered in the most recent deployment step P_4.

Conceptually, *Patch filtering* identifies the program variants that a patch impacts and could potentially influence their semantics. For this it is crucial to establish a clear definition of what it means for a variant to be "impacted" by a change [5]. Since determining the behavioral equivalence of two programs would also solve the halting problem, we must over-approximate. For compiled languages, *binary equivalence* can serve as a simple but safe heuristic: If a change does not affect a variant's resulting binary, it can be safely assumed that the change had no impact on that specific variant. However, it is important to note that not every change in the binary will result in a changed program semantic (e.g., linking order).

Nevertheless, binary equivalence presents three challenges for system integrators: (1) it requires the build process to be bit-wise reproducible, (2) it necessitates executing the entire build process, (3) for each shipped variant and every incoming change. As a result, binary equivalence can only be employed for variability-aware patch filtering when managing a few variants. With a larger number of variants, the approach is no longer feasible. For example, the Linux kernel is updated up to every five minutes [45], making it costly to build every version for even a single variant. Therefore, a faster and less resource-intensive alternative is desirable.

3 SIB: FAST VARIABILITY-AWARE PATCH FILTERING

We propose SiB, a fast patch filtering method for statically-configured software (see Fig. 1). For incoming patches, SiB consults its *line-range database (LRDB)* to determine the set of shipped variants that are impacted by the patch and might require further downstream actions. SiB is an over-approximating change-impact heuristic that relies on CPP-included source-code lines.

3.1 Line-Range DB

The SiB approach hooks into the life cycle of a statically-configured program at two points (Fig. 1): At variant deployment time, we collect information about every shipped variant, which we later use in the patch-filtering stage. While deployment is (and should be) a rare event, the patch-filtering process is triggered often, making overheads at this stage more critical than during deployment. Therefore, the choice of the knowledge base connecting both phases is a crucial decision for any multi-variant patch-filtering method.

In SiB we use covered source-code line numbers for this purpose and employ the *line-range database (LRDB)* to collect information about the shipped variants. In our example in Fig. 1, the LRDB knows that variant 1 used the source lines 1-2 and 6 of `foo.c`, while variant 2 included the range 43-50 of the header file `baz.h`. Using line ranges for the LRDB has multiple benefits: (1) Numerical ranges can be stored compactly, manipulated quickly, and specialized data structures (e.g., interval trees) exist to speed up LRDB queries in the patch-filtering stage. (2) Line numbers are a language-agnostic method to address parts of the source code, allowing us to support different annotation-based static-variability mechanisms (e.g., `#ifdef` and Rust's `#[cfg]`). (3) The unified patch format also relies on line numbers to anchor textual changes to the source code.

In our implementation we use a textual on-disk format to store the LRDB and construct an in-memory representation for the approval phase. For line-range overlap detection, we use the Rust `lapper` module, which implements the BITS [18] algorithm.

3.2 CPP Line Ranges

For C/C++ programs, the CPP is the standard tool that implements conditional compilation. For each compilation unit, the CPP receives the token stream from the lexer and interprets three kinds of directives: (1) macro definitions and expansions, (2) inclusion of other files, and (3) conditional in/exclusion of CPP blocks. As deselected CPP blocks have no impact on the compilation process, we can exclude their line ranges for the currently compiled variant.

To extract the relevant line ranges for a specific variant, we hook into the compilation process, which is always required for the deployment stage. For this purpose we built a Clang compiler plugin that uses the `PPCallbacks` interface to record the file ranges that the preprocessor feeds to the C parser. The plugin consists of 250 lines of code that mainly set up the integration with the Clang compiler. Thereby, we do not only cover the main source file but also all lines and conditional blocks that get included. After all files are compiled, we consolidate and merge the line ranges per source file, particularly for header files, into (filename → line-range) pairs.

Special attention is required for the lines containing CPP directives: For the used ranges, we always have to include the lines that mark the beginning (e.g., `#ifdef`) and the end (e.g., `#endif`) of a conditional block. Further, as CPP allows building `#if-#elif-#else` chains, we have to include the lines of all evaluated CPP directives in the set of important lines. After all, only the negative evaluation of the condition in line 1 (see Fig. 1) results in the inclusion of block B into variant 2. Therefore, a change to line 1 also impacts variant 2.

Further, we must handle changes to the build system that impact compiler flags influencing the compilation process. To do this, we rely on the "compilation database" [42], which lists all compiler invocations during the build process. Various build systems, e.g., CMake and Bazel, already produce this database as it facilitates integration with different tools and IDEs. By using the compilation database, we also cover build-system variability [10], as we know which compilation units are included in which variant. Whenever the database changes, we must assume changed compilation output.

Moreover, we must handle files affecting the compilation without them being used by the compiler. These files can be divided into two categories. (1) Files that are tracked by the version control systems and (2) files that are not:

For (1), the most common case is assembly files, which are usually compiled by an assembler and, therefore, bypass our compiler plugin. While we detect assembly files automatically, other files in this category must be passed to our tool by the developer. When the version control system signals a change in one of those files, we have to assume a patch is relevant.

Category (2) files are typically generated during the build process (e.g., parsers or stub/skeleton code). One way to handle these is to pass the generator input and, if necessary, the generator itself to our tool as described for category (1). However, these files could be configuration-dependent. At the developers' discretion, we therefore detect changes by storing the file hashes, which requires the files' existence before executing our tool. Whenever a hash changes, we assume the patch is relevant. While the developer may have domain-specific knowledge allowing better decisions, this process can be partially automated by comparing files in the compilation database with files processed by our compiler plugin and files known to the version control system.

We store the line-range information in the LRDB. For each deployment, we can either replace an existing shipped variant or introduce a new one. By storing line ranges, we have a compact and summarized over-approximation of the shipped variants that is used to identify irrelevant patches in the next step.

3.3 Line-Range–Based Patch Filtering

With the LRDB, we can now identify patches that have undoubtedly no influence on one or multiple shipped variants. Essentially, we extract the line ranges that the patch modifies and search the LRDB for variants with ranges that overlap with the patched ranges.

The unified patch format (see Fig. 1) is a line-based format containing multiple *hunks* of textual differences between two text files. Each hunk consists of a tuple (filename, start line, hunk length) for both the "source" and the "destination" file. For our concerns, the source file corresponds to a file that was (or could have been) used for a shipped variant. In our example, the patch describes a change to the file `foo.c` in line range 1-3 for both source and destination. Further, the patch data's first column indicates the modification: lines starting with a minus represent a line removal, while plus signs indicate line additions. Lines that start with a space remain unchanged and are only included in the patch as context to allow the patch application to modified source files.

For SiB, we use the version control system (i.e., `git`) to calculate a patch for the change that does *not* include context lines. By doing so, we focus on the actual changes and can accurately extract the modified line ranges from the hunk's metadata by examining the *source* line range. As the source file corresponds to a file in a shipped variant, we extract those as the patched ranges.

Special attention must be given to two corner cases: If the source and destination file names in a hunk do not match or if a hunk introduces or removes an *unbalanced* CPP expression, we mark the entire source file as changed. For example, if a patch introduces a single `#else`-directive after line 4 in Fig. 1, line 5 would be included in a variant with `CONF_A=0` and `CONF_B=0`, even though it was previously located in an excluded CPP block. To capture situations where a local change modifies the CPP-program structure, we parse each hunk's patch data and count block beginnings and closings for both source and destination separately. If one of those counters drops below zero, we detect a hunk with an unbalanced CPP structure and mark the entire source file as changed. We can limit this fallback to unbalanced hunks, as the introduction or removal of a balanced CPP structure into an excluded CPP block has no impact on subsequent CPP operations.

After extracting the modified line ranges from the patch, we can query the LRDB. If we find a range overlap, the identified variant *could* be impacted by the patch, which should then trigger further downstream actions (e.g., redeployment). Conversely, if we do not find an overlap, the patch cannot have any impact, as it only touched source lines that were irrelevant for the shipped variant(s).

4 EVALUATION

In our evaluation, we demonstrate that SiB only introduces a small processing overhead, while effectively filtering out irrelevant

patches for specific variants. For this, we analyze the development history of OSS projects that use static variability and further examine SiB's effectiveness with real-world security fixes for OpenSSL.

4.1 Case Studies

We focus our evaluation on four open-source projects: Linux, OpenSSL, SQLite, and Bochs. We selected these projects because of their widespread adoption and because they cover a wide range of software projects – an operating system, a cryptographic library, an embeddable database engine, and an emulator. Furthermore, they are written in the C/C++ programming language and use CPP-managed static variability.

Set of Patches To analyze different patch types (bug fixes, security improvements, and new features), we examine 200 consecutive commits from each project's development history. Each commit is treated as a separate patch that triggers the patch-management process and becomes subject to patch filtering. For our analysis, we discard defective commits (i.e., failed builds). For OpenSSL, SQLite, and Bochs, defective commits account for 3.36, 6.91, and 0.97 percent, respectively, while there are no defective commits for Linux. Whenever we compare two commits and one of them is defective, we exclude that comparison from our analysis.

Linux is a general-purpose operating system that is often used in embedded systems and a widely-studied SPL [7, 41]. We use Linux version 5.19 (and 200 preceding commits) with 15 configurations, including `defconfig`, `tinyconfig`, and 13 other random configurations. To generate the random configurations, we utilized the make target `randconfig` and set `KCONFIG_PROBABILITY=10` to control the variant size. The `defconfig` variant includes about 10 percent of the source code, comprising Assembly, C, and header files.

OpenSSL is a library that provides cryptographic primitives and TLS/SSL-secured connections. We use OpenSSL version 3.0.5 (and 200 preceding commits) with the default configuration and 14 random configurations. We generated these random configurations by passing random sets of feature flags, chosen from a pool of 95 total flags, to the configuration script. The default configuration includes 95 percent of the code.

SQLite is a serverless and lightweight database engine that is commonly used in embedded systems. We use SQLite version 3.37.2 (and 200 preceding commits) with the default configuration and 14 random configurations. The configurations were generated by passing random sets of feature flags out of a total of 12 flags to the configuration script. Further, we had to disable SQLite's "Amalgamation" process, which concatenates all source files for the compilation, as Amalgamation disconnects the line ranges of a patch from those line ranges the compiler sees. With the default configuration, 60 percent of the code is included.

Bochs is an x86 emulator commonly used for operating system development. In contrast to the other projects, it is written in C++. We use Bochs version 2.8 (and 200 preceding commits) with the default configuration and 14 random configurations. We generated these random configurations by passing random sets of feature flags, chosen from a pool of 57 total flags, to the configuration script. The default configuration includes 37 percent of the code.

Further Adaptations We had to adjust the source code of our projects to make their build process fully reproducible (see also Sec. 2.3) and compatible with binary-equivalence filtering. For example, we had to deselect certain configuration options [43] in Linux and disable the CPP `__LINE__` macro. Please note that these modifications were made solely for evaluation and validation purposes (i.e., to use binary equivalence as ground truth, which requires reproducible builds). They are *not* necessary when using SiB. Furthermore, since OpenSSL, SQLite, and Bochs do not yet employ a compilation database, we used the `compiledb`[1] tool to extract this information.

Table 1: BE-Based vs. SiB – Costs per Variant

[s]	BE-Based				SiB			
Step	Linux	OpenSSL	SQLite	Bochs	Linux	OpenSSL	SQLite	Bochs
Build[a] (*B*)	23.36	12.39	26.2	22.68	23.53	12.43	26.46	22.83
Store[a] (*S*)	0.35	0.12	0.04	0.05	0.99	0.23	0.09	0.1
Check (*C*)	0.004	0.004	0.001	0.001	0.062	0.023	0.005	0.005

[a]: SiB requires these steps only for patches that trigger a deployment.

4.2 Evaluation Scenario

To evaluate SiB, we build the selected 200 commits of our target projects (Linux, OpenSSL, SQLite, Bochs) with 15 different configurations each. For each commit we use the predecessor commit for the shipped variants, while the current commit acts as the patch we want to filter.

Our ground truth is binary equivalence, that is, we assume that a patch is relevant for a specific configuration if and only if the output binary changes. A patch not introducing a change cannot have any effect and, therefore, can safely be classified as not relevant. To eliminate linker-induced indeterminism, we use the linker input-file hashes to assess binary equivalence instead of comparing the final binaries. We refer to this approach as binary-equivalence-based (BE-based). While this approach represents a significant improvement over manual and no patch filtering, it is slow and scales poorly with the number of variants. As we expect SiB to run as part of a continuous integration process, we execute our evaluation on a large server machine with two 26-core Intel® Xeon® Gold 5320 @ 2.20 GHz (104 hardware threads) and 256 GB of memory. We compiled with Clang 15 on Ubuntu 23.10. The BE-based approach resulted in over 12 000 builds and took nearly three days to complete on our high-performance server. As we consider binary equivalence our ground truth, the BE-based approach trivially yields perfect accuracy.

4.3 Patch-Filtering Costs

First, we want to answer the question (RQ1), *if and under which circumstances SiB outperforms the BE-based approach?* The patch-filter costs for SiB consist of two components: (1) Deployment overheads for extracting and processing used line ranges and (2) the time required to filter an incoming patch. Unless stated otherwise, all numbers are the median of all commits and variants.

Deployment Overheads When shipping a variant, we have to build a binary, during which SiB's compiler plugin extracts the

[1] https://github.com/nickdiego/compiledb

used lines for the current configuration, processes them, and stores them in the LRDB. In comparison, the BE-based approach adds no overhead to the build process itself but needs to calculate and store the object-file hashes (see Tab. 1). For example, building OpenSSL usually takes 12.39 s, which increases with SiB to 12.43 s (+0.32 %). The build overheads for Linux, SQLite, and Bochs are 0.71, 0.99, and 0.64 percent. For calculating and processing the object hashes, the BE-based approach takes between 0.04 s and 0.35 s, while SiB requires between 0.09 s to 0.99 s. While all information can be summarized into a single hash when using the BE-based approach, SiB requires data for each configuration and version, which on average sums up to 487 KiB, 164 KiB, 88 KiB, and 107 KiB for Linux, OpenSSL, SQLite, and Bochs, respectively. For a patch that triggers a deployment, the BE-based approach adds 1.5 percent (0.35 s) at most, while SiB's overhead is at most 5 percent (1.16 s) at most.

Filtering Overheads To filter an incoming patch, SiB must only match the changed lines against its LRDB, while the BE-based approach must (re-)build the project for every variant, collect and store object hashes, and compare them. Therefore, SiB will usually outperform the BE-based approach, although hash comparison (< 0.004 s) is much faster than LRDB lookups (0.062 s for Linux). As this trade-off decides upon the *patch-impact probability*, we define a cutoff probability α ($\in [0, 1]$) so that if the probability that a patch impacts a variant is less than α, SiB will outperform the BE-based approach. With the information in Tab. 1, we can set up Equation 1:

$$B_{\text{BE}} + S_{\text{BE}} + C_{\text{BE}} = C_{\text{SiB}} + \alpha \cdot (B_{\text{SiB}} + S_{\text{SiB}}) \tag{1}$$

The fixed costs for a new patch of the BE-based approach are compared to the costs of SiB depending on α. When using SiB and if the patch impacts a variant, in addition to checking, there are also the costs of building a new release and storing the information in the LRDB. Solving for α leads to Equation 2:

$$\alpha = \frac{B_{\text{BE}} + S_{\text{BE}} + C_{\text{BE}} - C_{\text{SiB}}}{B_{\text{SiB}} + S_{\text{SiB}}} \tag{2}$$

With our current implementation, the tipping points are at 96.4/98.7/98.8/99.1 percent for Linux/OpenSSL/SQLite/Bochs. However, if a customer faces such high α values, patch-filtering almost has no impact on the patch-management process anyways. Please also note that SiB could be further optimized (e.g., LRDB with lookup indices) to bring α even closer to one.

Another build-step optimization would be the usage of CCache [35], a cache that reduces compiler invocations by associating object files with the hash over the preprocessed source code. If applied, we see a build speedup of 24.1/2.6/5.5/5.8 percent (Linux/OpenSSL/SQLite/Bochs) for the BE-based approach. However, even with a 50 percent build speedup, the α tipping points for SiB only decrease to 93.2/98.5/98.3/99.0 percent.

Summarized, we can answer RQ1 and conclude that SiB is more efficient than a binary-equivalence-based approach if the patch-impact probability is less than 95 percent.

4.4 Patch-Filtering Accuracy

We now want to look at the accuracy of SiB and raise three research questions: (RQ2) *Is the accuracy of SiB comparable to the BE-based approach?* (RQ3) *Is patch-filtering reducing shipment costs?* (RQ4) *Is the patch-management process sensitive to variability?* To answer these questions, we take the result of the BE-based approach as ground truth and measure how often SiB deviates from it.

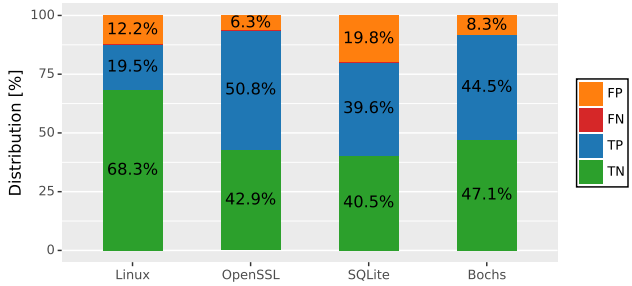


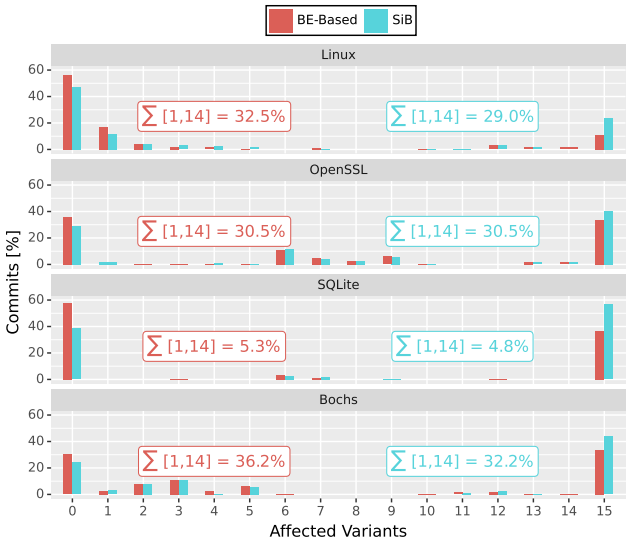
Figure 3: Accuracy of SiB. The impact detection of SiB has been classified as true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) in comparison to binary equivalence for the four target projects – Linux, OpenSSL, SQLite, and Bochs. Notably, SiB has achieved zero false negatives, meaning no changes have gone undetected.

In Fig. 3 we summarize the accuracy of SiB for the projects: The absence of false negatives, and therefore a recall of 100 percent for all evaluated projects, implies that SiB does not filter out any patch that binary equivalence deemed to be important, making SiB a safe patch-filtering method.

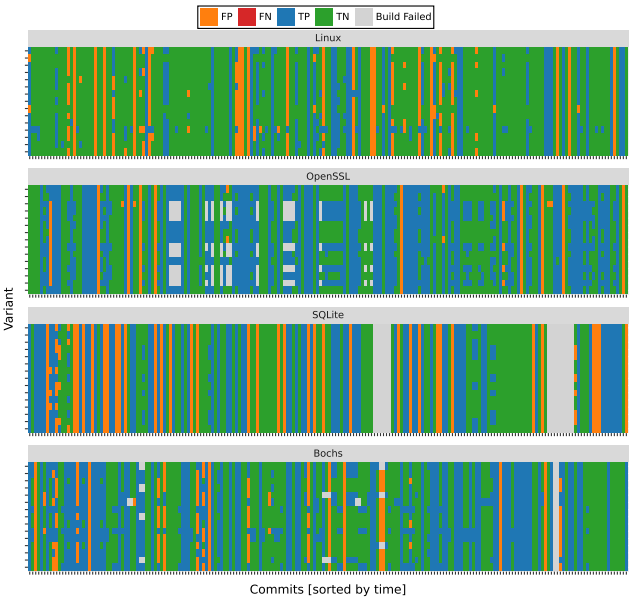
In contrast, false positives, which means that a patch is falsely assumed to affect a given variant, are less critical and only impact the efficiency. The observed FP rates between 6.31 percent (OpenSSL) and 19.82 percent (SQLite) stem from three causes: (1) Purely syntactical changes (e.g., changing a comment) to a selected CPP block trigger SiB but do not alter the binary. (2) When detecting changes on the CPP level, SiB cannot ignore semantic changes in unused type declarations. (3) SiB takes a conservative approach concerning changed CPP directives (see Sec. 3.2). It would be possible to reduce the number of false positives, for example, by discarding comments. More advanced techniques require leaving the CPP level and, therefore, would likely be performed for each variant independently, contradicting SiB's philosophy. Nevertheless, even with the observed FP rate, the positive rate (FP+TP) remains below the calculated tipping points α. By looking at the accuracy (TN+TP) of SiB, which is 88/94/80/92 percent and the precision, which is 62/89/67/84 percent (Linux/OpenSSL/SQLite/Bochs), we can answer RQ2 positively.

The BE-based approach reduces shipments by 49 to 80 percent, while SiB achieves 41 to 68 percent (see Fig. 3), which answers RQ3: Patch filtering is able to reduce the number of variant shipments, significantly reducing the overall patch-management costs.

However, it still might be the case that real changes are not variant-sensitive, meaning a patch either impacts no or all variants. To investigate this, for each commit we count the number of variants that it impacts. In Fig. 4a we see that in Linux, OpenSSL, and Bochs around 30 percent of all commits impact between 1 and 14 variants and that SiB captures these as effectively as the BE-based approach. For SQLite, however, patch management is not very specific to the chosen variant, as only 5 percent of patches impact only some variants.



(a) Histogram. The number of variants affected by which percentage of commits for Linux, OpenSSL, SQLite, and Bochs (BE-based and SiB).



(b) Classification Map. Each variant for every commit classified as true positive (TP), true negative (TN), false positive (FP), false negative (FN), and build failure compared to binary equivalence for Linux, OpenSSL, SQLite, and Bochs.

Figure 4: Affected Variants

In Fig. 4b, a classification map for our projects, this property of SQLite is reflected by the uniform horizontal bars. A less regular map (only TPs) signifies a high sensitivity of a project's patch-management process with regard to variability. As expected, Linux,

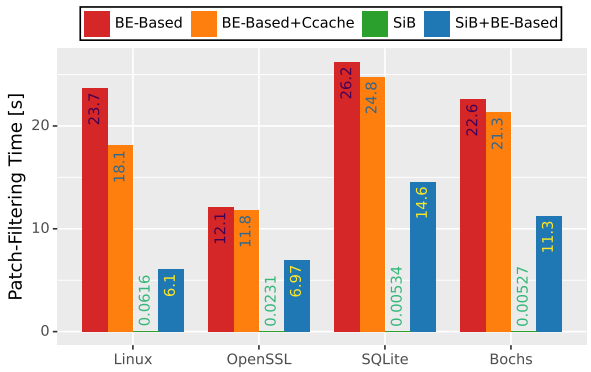


Figure 5: Patch-Filter Overhead. The mean time required to filter a new patch for Linux, OpenSSL, SQLite, and Bochs with 15 variants.

a commonly used example for SPLs with a large feature model (> 17 000 features [22]), shows an irregular classification map.

With these insights we can answer RQ4: For projects developed as SPL, the patch-management process is sensitive to variability. Therefore, a variability-aware patch-filtering approach is required.

Hybrid Patch Filtering As both the BE-based approach and SiB have their benefits (accuracy and efficiency, respectively), the question arises as to how a *hybrid* patch-filtering method performs: For each incoming patch, we first run SiB to narrow down the number of variants, harvesting SiB's efficiency, before then building only the remaining variants and applying BE-based approach, utilizing the accuracy of binary equivalence. In Fig. 5 we compare the patch-filtering costs for the BE-based approach (with and without CCache), SiB, and the hybrid approach. In total, the hybrid approach results in a filtering-time reduction of 74 percent (Linux), 42 percent (OpenSSL), 44 percent (SQLite), and 50 percent (Bochs) compared to the BE-based approach.

Also, the additional costs for hybrid patch filtering are not lost if we can reuse the resulting binary in the process further downstream. However, please note that the actual benefit of patch filtering is not to avoid or reduce build costs but to prune branches of the patch-management process (i.e., assessment, prioritization, testing, deployment), which are considerably more costly and often involve manual intervention at some point.

4.5 High-Priority Patches

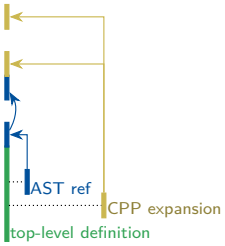
We have already demonstrated the effectiveness of SiB in filtering patches, which could be especially rewarding where the timely application of patches is crucial and waiting for an opportune moment is not feasible. As discussed in Sec. 1, the application of high-priority patches (like Heartbleed) is commonly assumed to be non-negotiable, which raises the question (RQ5) of *whether these patches are also configuration-related.*

To investigate this, we looked at the Heartbleed patch and also searched in our evaluated commit range and found three further CVE-related OpenSSL patches: The first, `CVE-2022-2097`, only affected 32-bit systems. The second, `CVE-2022-2068`, affected a shell script and, thus, was irrelevant to our evaluation. The third, `CVE-2022-29242`, was a security vulnerability in the GOST engine.

Table 2: LRDB Coverage (default configuration)

		Linux		OpenSSL		SQLite	
		Files	Lines	Files	Lines	Files	Lines
Repository	Total	55 795	30 649 128	1 792	632 017	394	389 918
	Code	–	74.79 %	–	77.08 %	–	69.31 %
	Comments	–	12.58 %	–	11.80 %	–	23.72 %
	Blank	–	12.63 %	–	11.12 %	–	6.97 %
int.	#included	6 031	3 206 310	1 753	599 016	126	235 213
	CPP-covered	6 031	94.8 %	1 753	95.5 %	126	62.1 %
	AST-covered	5 752	72.1 %	1 720	80.3 %	92	45.0 %
ext.	#included	–	–	263	39 093	149	20 904
	CPP-covered	–	–	263	86.1 %	149	82.5 %
	AST-covered	–	–	125	6.1 %	51	5.1 %

```
1 #define FOO(X)   ((X) + 1)
2 #define BAR(Z)   CONSTANT
3 #define CONSTANT 23
4 struct foo { int x; };
5 struct bar { struct foo f; };
6 struct baz { struct foo f; };
7 int main() {
8     struct baz obj;
9     obj.f.x = FOO(CONSTANT);
10 }
```



Listing 2: AST-Level Line Coverage

For all these CVE commits, along with the Heartbleed patch, SiB accurately narrowed down the set of affected variants to configurations that were genuinely impacted by the changes; all other configurations were marked as unaffected. Since we use random configurations, except for the default configuration, no statement about the actual number of variants affected in the real world can be made. Still, we conclude that even the expensive high-priority patches can be filtered out using SiB (RQ5).

4.6 AST-Level LRDB Information

So far we have shown that SiB is capable of filtering out irrelevant patches. Central to our method is the LRDB, which we populate with line ranges that the *C preprocessor (CPP)* feeds to the C parser. While we have already quantified (in Sec. 4.3) that gathering information at the CPP level is fast, its granularity is coarse and results in an LRDB containing more line ranges than necessary. This brings us to our sixth research question (RQ6): *Is the CPP the appropriate level of abstraction for gathering LRDB information?*

To address this question, we propose a second method to populate the LRDB with information from the *abstract syntax tree (AST)* level. While we have implemented this AST-level LRDB extraction, we have not yet integrated it with our SiB approach. Our prototype is currently rather slow, only works on C source code, which is why Bochs is excluded, and would necessitate additional syntax-aware handling of incoming patches. However, it is already sufficient to answer RQ6 by comparing its results with the coverage ratio of the LRDB extracted at the CPP level.

The core idea of AST-level LRDB extraction is to record source-code line ranges that contain top-level definitions (e.g., global variables, function bodies). However, since these definitions are influenced by (1) top-level declarations (i.e., types, enums, function prototypes) and (2) CPP macros, we must follow these cross-references from the definition ranges recursively. This cross-reference search on the AST from top-level definitions draws inspiration from cHash [9], which calculates a hash value for each AST node. For LRDB extraction, we extended it conceptually by macro-expansion tracking. Again, we implemented this approach as a Clang plugin.

To make AST-level line coverage more intuitive, Lst. 2 provides a small example: The AST traversal starts at the `main()` function (lines 7-10). As `main()` uses the `struct baz` type, which itself references `struct foo`, we also include lines 4 and 6 in the LRDB. Since the declaration on line 5 is never referenced, we can exclude it. Similarly, the macros `FOO` and `CONSTANT` are used within an already used range, requiring us to include lines 1 and 3 in the LRDB, while the macro on line 2 is not referenced and can be excluded. If combined with the SiB approach, we could filter out patches that affect lines 2 and/or 5. In contrast, the CPP-only approach would include all lines in the LRDB, as there are no `#ifdef` statements present.

Coming back to RQ6, we execute both extraction methods (on CPP and AST level) for the standard configuration of our three open-source projects developed in C and quantify the number of source files and lines included in the LRDB (see Tab. 2). Further, we distinguish between lines that originate from within the project repository and those lines that stem from external headers.

For each project, the #included line indicates the number of files (and their accumulated source lines) that were selected by the build system. Due to its extensive SPL nature, the Linux standard configuration only includes around 10 percent of its source lines. For OpenSSL and SQLite, the gap between the repository and the included numbers mainly stems from test cases.

As expected, there is a considerable gap between the LRDB size if filled from the CPP or the AST level. For Linux and OpenSSL, about five percent of all lines are in excluded CPP blocks, while the AST level excluded four to five times as many lines. This gap is not only caused by the increased precision of the AST-level reachability analysis (i.e., unused type declarations) but also by lines outside a top-level definition that are blank or only contain a comment. For SQLite, which manages less of its variability using the build system, both CPP and AST level result in much smaller LRDBs.

In examining source code pulled from outside the source repository (i.e., headers), the AST approach substantially reduces the covered maintenance surface, which drops to less than 7 percent for OpenSSL and SQLite (Linux does not rely on external headers). Although headers, which are primarily comprised of type declarations, function prototypes, and macros, were anticipated to be covered to a lesser extent, the magnitude of this effect exceeded our initial expectations. This highlights the potential benefits of an AST-based approach to capture (configurable) build-time dependencies between different projects.

This provides us with a *partial* answer to RQ6: AST-level line ranges, as compared to those at the CPP level, will likely lead to a reduction in LRDB size. We anticipate this will correspondingly improve the effectiveness of patch filtering. However, it's important to bear in mind that this could be offset by increased overheads and additional challenges, which we will discuss in Sec. 5.1. Whether these potential drawbacks justify the use of an AST-based approach remains a subject for future research.

4.7 Evaluation Summary

Our evaluation demonstrates that SiB outperforms binary-equivalence-based patch filtering in terms of efficiency (RQ1) while maintaining comparable accuracy (RQ2). Furthermore, we establish that patches in real-world projects are indeed sensitive to variability (RQ4) and that SiB effectively filters out high-priority patches (RQ5). Additionally, we demonstrated (RQ6) that while populating the LRDB with CPP-level information is adequate, it could be further improved, especially if external headers should be considered, by utilizing AST-level information. But most importantly, we show that SiB can reduce deployments by up to 68 percent (RQ3).

5 DISCUSSION

Several potential threats could impact the validity of our experimental results. The potential savings and, to a lesser degree, the overheads of SiB depend on (1) the configurations utilized, (2) the patches applied to the software, and (3) the extent of conditional compilation employed by the software. We have endeavored to minimize these threats by implementing a rigorous experimental design, which includes (1) the utilization of 15 distinct configurations, some of which were chosen randomly, (2) the evaluation of 200 patches per project that were not subjectively selected, and (3) the selection of four distinct target applications, each with different characteristics and complexity.

5.1 Applicability and Generalizability

SiB can be applied to any project that utilizes the CPP for managing variability. Although the extraction of relevant lines through the Clang plugin relies on the project's ability to be compiled using Clang, and the utilization of git for identifying changed lines requires a git repository, the underlying concept is not limited to a specific compiler or VCS. Even the CPP could be replaced by other preprocessors like M4. The dependency on a compilation database is easily addressed through the use of external tools, rendering it a non-issue. This versatility is further bolstered by SiB's resilience to build instability. As long the instability has no semantic impact, SiB also integrates, in contrast to binary equivalence, with a build process that does not produce bit-wise–identical artifacts.

The line-based concept itself offers potential for generalization to other languages and variability mechanisms. Essentially, such a generalization would require two key steps: (1) identifying which lines are relevant for a given variant and (2) determining whether we are allowed to match a patch against the LRDB.

As we move towards semantically richer variability mechanisms, these steps become more complex than for CPP-based variability. For example, in Sec. 4.6, we illustrated the first step (filling the LRDB) for the C programming language, which involves a static reachability and usage analysis on the AST. However, for this variability mechanism, we have not yet addressed step (2). The principal question posed to a patch is whether it impacts the variability structure itself. For CPP this is simple as it suffices to check it for a new CPP block. However, for C-based variability we would already have to check whether a patch introduces or splits a function.

For more complex languages such as C++, this step (2) becomes even more challenging: A patch can introduce a new function that influences the overload resolution for existing function lookups. If the patch performs its modifications in a line range not included in the variant, we would encounter a false negative.

On the other hand, operating at the AST level would further enable us to encompass statically-evaluated dynamic variability: Developers use configuration switches to deliberately enable or disable certain control-flow branches (i.e., `if (CONF_A)...`), which are subject to standard type checking, unlike disabled CPP blocks. If disabled, the compiler's dead-code elimination removes the dead then-branch entirely. For our AST-level LRDB, we would exclude such disabled code blocks if we were certain that their guarding condition can never evaluate to true.

So while the line-based approach is highly generalizable, we believe we have found a suitable implementation. However, it comes with constraints that might hinder its applicability: (1) We extract line ranges for all shipped variants. Consequently, SiB's costs scale with the number of shipped variants. While it will scale much better than the BE-based approach, situations with enormous variant sets may limit its benefits. (2) Although we do not require a semantic understanding of the patch, we still have to be conservative with changes to the variability structure itself. While these, if occurring often, could deteriorate SiB's benefits, studies of CPP usage [19] showed that developers tend to use it in a "disciplined" fashion.

5.2 Further Usage in the Development Process

Patch management is not confined to the maintenance phase but is also part of the development cycle before the software is shipped to the customer. In this context SiB could also be employed in the domain of regression test selection (RTS) [36] for SPLs. RTS answers the question of which downstream (unit) test cases need to be (re-)executed after a certain modification is made. By integrating SiB into this process, we could effectively select only the relevant test cases that are impacted by the change.

SiB could also be used to provide developers with real-time feedback from their integrated development environment (IDE). As LRDB queries, even in large projects like Linux, are executed swiftly, developers can receive immediate feedback regarding the impact of their intended changes. Thereby, developers are directly informed which shipped variants and, consequently, which customers would be affected by a planned change. This knowledge can aid developers in exercising caution, particularly when a large number of customers stand to be impacted.

The LRDB's information may be utilized to find source code not used by *any* shipped variant. This could help with debloating variability, where tool support has been reported as sparse [1].

6 RELATED WORK

The challenge we address resides at the intersection of two key research areas: patch assessment and static analysis of *configurable* software. As most related works tend to focus on only one area, we first discuss these areas separately before exploring their overlaps.

Patch Assessment Considering the vast array of research on patch assessment, we will focus on a select few that are notably relevant to our work.

Our patch filtering technique is closely related to change-impact analyses [5], which aim to automatically understand the impact of

a change. However, in contrast to such detailed analyses, SiB only provides a binary impact decision, thus promoting its efficiency.

Extensive research has been conducted in the realm of patch classification: Sawadogo et al. [38] classify patches based on their relevance to security, whereas Murgia et al. [30] offer more nuanced classifications by parsing commit messages. Lomio et al. [23] have worked on identifying patches that could potentially introduce new security vulnerabilities. Another facet of this domain is patch correctness – determining whether a patch is safe to apply without altering the program's intended behavior, as is often the case with security patches [24]. Wang et al. [48] offer a comprehensive survey on the assessment of patch correctness, specifically for patches generated by automated systems. Patch retrieval (see Fig. 2) is the process of identifying whether a software system is vulnerable to a specific vulnerability. In this context Plate et al. [31] examined known vulnerabilities in executed code, including dependencies.

Although these approaches address more complex problems than SiB, which solely determines whether a patch influences a program variant, they do not account for statically-configured software. Thus, these intricate, and hence slower, analyses must be executed for each configuration, leading to inefficient scaling behavior. To our knowledge no research has been conducted on the binary decision of patch relevance for statically-configurable software.

Static Analysis of Configurable Software Configurable software poses a significant challenge for static analysis due to the co-existence of multiple variants in a single code base.

Sampling [46] is one strategy often employed to mitigate this issue by individually analyzing a large number of (random) configurations, aiming to cover the entire configuration space. SiB complements such methods if we treat each sampled configuration as a shipped variant. While sampling provides a level of coverage, it lacks soundness. To ensure comprehensive coverage of the entire SPL, variability-aware static analysis was proposed. Such methods often involve configuration-aware parsing as done by tools like SuperC [13] or TypeChef [14]. For example, Liebig et al. [20] utilize TypeChef to propose variability-aware type-checking and liveness analyses. However, configuration-aware parsing can be slow for larger projects and relies on the actual programming language (i.e., C), a dependency that SiB does not have.

Maintaining configurable software is another critical aspect: Vamypr [40] is a tool designed to provoke variability-dependent compiler warnings. Further, Angerer et al. [3] propose a dead code analysis, although it only applies to the current configuration. Undertaker [41] extends this concept, translating the feature model into propositional logic to identify dead CPP blocks across all possible configurations. Zhang and Becker [50] present metrics for variability models that help improve the SPL, while Lillack et al. [21] analyze load-time configuration usage in Android apps.

While variability-aware static analysis provides a range of methods, these are typically designed with variability in mind and often require specialized knowledge about the software (e.g., variability model). Unlike these tools, SiB does not require this specialized information and it targets the later stages of the software life cycle, as opposed to focusing on the development phase.

Variability-Aware Change Impact Few works have integrated the field of change-impact analysis with variability-aware program analysis. Angerer et al. [2] propose a configuration-aware change impact analysis for load-time variability, using an interprocedural and conditional system dependency graph. While their approach is elegant, it relies on extracting configuration-aware dependencies from the AST. As we target compile-time variability, this would require a variability-aware parser, like TypeChef or SuperC. Wang et al. [47] undertook such a complete approach based on variability-aware functional-equivalence checking. Their approach takes seconds for projects with few features and requires a formal feature model, which is often not available for OSS components. Further, the analyzed program must not contain global variables, jumps, or recursive function calls. Instead of identifying all possible configurations with changed semantic, SiB restricts itself to the known and, therefore, relevant configurations, enabling faster results. Test2Feature [25] handles regression test selection. Depending on a change, only tests regarding affected features should be executed. Michelon et al. [27] analyze and propagate feature revisions. While both approaches, like SiB, work on line-based textual changes, mapping features to lines is handled as a constraint satisfaction problem, which differs fundamentally from SiB. Schwarzkopf et al. [39] address a conceptually similar problem in the context of virtual machine images, but there are no technical similarities. Their approach involves performing checks for updated software packages across all images in a cloud environment.

Our proposal, SiB, bridges the gap between patch management and configurable-software research. By filtering out irrelevant patch-variant combinations, SiB can reduce the number of variants that need to be processed by the patch-management process, all without requiring fundamental modifications to the process.

7 CONCLUSION

Patch management and deciding on patch roll-outs (e.g., for security fixes) is a tedious and expensive process in software maintenance. However, most system software is highly compile-time configurable, so there is a high chance that (even critical) patches do not affect all shipped or installed variants.

Therefore, we proposed variant-aware patch-filtering as a part of the early patch assessment process for statically-configured software with a predefined set of variants. Our SiB approach integrates with the C preprocessor during build time to record the relevant line-number ranges for each variant, which are matched against incoming patches to determine the set of affected variants.

In our evaluation, conducted on four large system-software projects (Linux, OpenSSL, SQLite, Bochs), we could confirm that for a set of 15 (random) configurations SiB correctly classifies up to 68 percent of variants as unaffected and, therefore, can reduce deployments significantly. Moreover, SiB does not produce false negatives, so it is also usable and effective to filter out high-priority security patches for unaffected variants.

Please refer to the published artifacts to verify and replicate the experiments [17].

ACKNOWLEDGMENTS

We would like to thank our reviewers for their constructive feedback. This work has been supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 236869097.

REFERENCES

[1] Mathieu Acher, Luc Lesoil, Georges Aaron Randrianaina, Xhevahire Tërnava, and Olivier Zendra. 2023. A Call for Removing Variability. In *17th Intl. Working Conf. on Variability Modelling of Software-Intensive Systems*. ACM. https://doi.org/10.1145/3571788.3571801

[2] Florian Angerer, Andreas Grimmer, Herbert Prahofer, and Paul Grunbacher. 2015. Configuration-Aware Change Impact Analysis (T). In *2015 30th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*. IEEE. https://doi.org/10.1109/ASE.2015.58

[3] Florian Angerer, Herbert Prähofer, Daniela Lettner, Andreas Grimmer, and Paul Grünbacher. 2014. Identifying inactive code in product lines with configuration-aware system dependence graphs. In *18th Intl. Software Product Line Conf. - Volume 1*. ACM. https://doi.org/10.1145/2648511.2648517

[4] Apache. 2021. *CVE - CVE-2021-44228*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-44228

[5] Robert S Arnold. 1996. *Software change impact analysis*. IEEE Computer Society Press.

[6] Maider Azanza, Leticia Montalvillo, and Oscar Díaz. 2021. 20 years of industrial experience at SPLC: a systematic mapping study. In *25th ACM Intl. Systems and Software Product Line Conf.*

[7] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Trans. on Software Engineering* 39, 12 (2013). https://doi.org/10.1109/TSE.2013.34

[8] Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley.

[9] Christian Dietrich, Valentin Rothberg, Ludwig Füracker, Andreas Ziegler, and Daniel Lohmann. 2017. cHash: Detection of Redundant Compilations via AST Hashing. In *2017 USENIX Annual Technical Conf. (USENIX '17)*. USENIX Association. https://www.usenix.org/conference/atc17/technical-sessions/presentation/dietrich

[10] Christian Dietrich, Reinhard Tartler, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. A Robust Approach for Variability Extraction from the Linux Build System. In *16th Software Product Line Conf. (SPLC '12)*, Eduardo Santana de Almeida, Christa Schwanninger, and David Benavides (Eds.). ACM Press. https://doi.org/10.1145/2362536.2362544

[11] Nesara Dissanayake, Asangi Jayatilaka, Mansooreh Zahedi, and M Ali Babar. 2022. Software security patch management-A systematic literature review of challenges, approaches, tools and practices. *Information and Software Technology* 144 (2022).

[12] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. 2014. The Matter of Heartbleed. In *2014 Conf. on Internet Measurement Conf. (IMC '14)*. Association for Computing Machinery. https://doi.org/10.1145/2663716.2663755

[13] Paul Gazzillo and Robert Grimm. 2012. SuperC: parsing all of C by taming the preprocessor. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '12)*. ACM Press. https://doi.org/10.1145/2254064.2254103

[14] 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 *26th ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. ACM Press. https://doi.org/10.1145/2048066.2048128

[15] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020).

[16] Hsuan-Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *ACM on Measurement and Analysis of Computing Systems* 4, 1, Article 03 (2020). https://doi.org/10.1145/3379469

[17] Tobias Landsberg, Christian Dietrich, and Daniel Lohmann. 2024. Should I Bother? Fast Patch Filtering for Statically-Configured Software Variants – Artifacts. https://doi.org/10.5281/zenodo.11611859

[18] Ryan M. Layer, Kevin Skadron, Gabriel Robins, Ira M. Hall, and Aaron R. Quinlan. 2012. Binary Interval Search: a scalable algorithm for counting interval intersections. *Bioinformatics* 29, 1 (2012). https://doi.org/10.1093/bioinformatics/bts652

[19] Jörg Liebig, Christian Kästner, and Sven Apel. 2011. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *10th Intl. Conf. on Aspect-Oriented Software Development (AOSD '11)*, Shigeru Chiba (Ed.). ACM Press. https://doi.org/10.1145/1960275.1960299

[20] Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. 2013. Scalable analysis of variable software. In *2013 9th Joint Meeting on Foundations of Software Engineering*. ACM. https://doi.org/10.1145/2491411.2491437

[21] Max Lillack, Christian Kästner, and Eric Bodden. 2014. Tracking load-time configuration options. In *29th ACM/IEEE Intl. Conf. on Automated Software Engineering*. ACM. https://doi.org/10.1145/2642937.2643001

[22] Daniel Lohmann. 2022. 'What is the Ideal Operating System?': Technical Perspective. *Commun. ACM* 65, 5 (2022). https://doi.org/10.1145/3524299

[23] Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. 2022. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software* 188 (2022). https://doi.org/10.1016/j.jss.2022.111283

[24] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. 2020. SPIDER: Enabling Fast Patch Propagation In Related Software Repositories. In *2020 IEEE Symp. on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/SP40000.2020.00038

[25] Willian D. F. Mendonça, Silvia R. Vergilio, Gabriela K. Michelon, Alexander Egyed, and Wesley K. G. Assunção. 2022. Test2Feature: feature-based test traceability tool for highly configurable software. In *26th ACM Intl. Systems and Software Product Line Conf. - Volume B*. ACM. https://doi.org/10.1145/3503229.3547031

[26] Neel Metha, Riku, Antii, and Matti. 2014. *Heartbleed Bug*. https://heartbleed.com/

[27] Gabriela K. Michelon, Wesley K. G. Assunção, Paul Grünbacher, and Alexander Egyed. 2023. Analysis and Propagation of Feature Revisions in Preprocessor-based Software Product Lines. In *2023 IEEE Intl. Conf. on Software Analysis, Evolution and Reengineering (SANER)*. https://doi.org/10.1109/SANER56733.2023.00035

[28] Gabriela K. Michelon, Wesley K. G. Assunção, David Obermann, Lukas Linsbauer, Paul Grünbacher, and Alexander Egyed. 2021. The life cycle of features in highly-configurable software systems evolving in space and time. In *20th ACM SIGPLAN Intl. Conf. on Generative Programming: Concepts and Experiences*. ACM. https://doi.org/10.1145/3486609.3487195

[29] Gabriela Karoline Michelon, David Obermann, Lukas Linsbauer, Wesley Klewerton G. Assunção, Paul Grünbacher, and Alexander Egyed. 2020. Locating feature revisions in software systems evolving in space and time. In *24th ACM Conf. on Systems and Software Product Line: Volume A - Volume A*. ACM. https://doi.org/10.1145/3382025.3414954

[30] Alessandro Murgia, Giulio Concas, Michele Marchesi, and Roberto Tonelli. 2010. A machine learning approach for text categorization of fixing-issue commits on CVS. In *2010 ACM-IEEE Intl. Symp. on Empirical Software Engineering and Measurement*. ACM. https://doi.org/10.1145/1852786.1852794

[31] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE Intl. Conf. on Software Maintenance and Evolution (ICSME)*. IEEE. https://doi.org/10.1109/ICSM.2015.7332492

[32] Matthew Prince. 2014. *The Hidden Costs of Heartbleed*. https://blog.cloudflare.com/the-hard-costs-of-heartbleed/

[33] Rick Rabiser, Klaus Schmid, Martin Becker, Goetz Botterweck, Matthias Galster, Iris Groher, and Danny Weyns. 2018. A Study and Comparison of Industrial vs. Academic Software Product Line Research Published at SPLC. In *22nd Intl. Systems and Software Product Line Conf. - Volume 1 (SPLC '18)*. Association for Computing Machinery. https://doi.org/10.1145/3233027.3233028

[34] Georges Aaron Randrianaina, Djamel Eddine Khelladi, Olivier Zendra, and Mathieu Acher. 2022. Towards incremental build of software configurations. In *ACM/IEEE 44th Intl. Conf. on Software Engineering: New Ideas and Emerging Results*. ACM. https://doi.org/10.1145/3510455.3512792

[35] Joel Rosdahl. 2010. *Ccache — Compiler cache*. https://ccache.dev

[36] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *IEEE Trans. Softw. Eng.* 22, 8 (1996). https://doi.org/10.1109/32.536955

[37] Andreas Ruprecht, Bernhard Heinloth, and Daniel Lohmann. 2014. Automatic Feature Selection in Large-Scale System-Software Product Lines. In *13th Intl. Conf. on Generative Programming and Component Engineering (GPCE '14)*, Matthew Flatt (Ed.). ACM Press. https://doi.org/10.1145/2658761.2658767

[38] Arthur D. Sawadogo, Tegawendé F. Bissyandé, Naouel Moha, Kevin Allix, Jacques Klein, Li Li, and Yves Le Traon. 2022. SSPCatcher: Learning to catch security patches. *Empirical Software Engineering* 27, 6 (2022). https://doi.org/10.1007/s10664-022-10168-9

[39] Roland Schwarzkopf, Matthias Schmidt, Christian Strack, and Bernd Freisleben. 2011. Checking Running and Dormant Virtual Machines for the Necessity of Security Updates in Cloud Environments. In *2011 IEEE Third Intl. Conf. on Cloud Computing Technology and Science*. IEEE. https://doi.org/10.1109/CloudCom.2011.40

[40] 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. In *2014 USENIX Annual Technical Conf. (USENIX '14)*. USENIX Association. https://www.usenix.org/conference/atc14/technical-sessions/presentation/tartler

[41] Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. 2011. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *ACM SIGOPS/EuroSys European Conf. on Computer Systems 2011 (EuroSys '11)*, Christoph M. Kirsch and Gernot Heiser (Eds.). ACM Press. https://doi.org/10.1145/1966445.1966451

[42] The Clang Team. 2023. *JSON Compilation Database Format Specification*. https://clang.llvm.org/docs/JSONCompilationDatabase.html

[43] The kernel development community. 2019. *The Linux Kernel — Reproducible builds.* https://www.kernel.org/doc/html/latest/kbuild/reproducible-builds.html

[44] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Survey* 47, 1, Article 6 (2014). https://doi.org/10.1145/2580950

[45] Thomas Thüm, Leopoldo Teixeira, Klaus Schmid, Eric Walkingshaw, Mukelabai Mukelabai, Mahsa Varshosaz, Goetz Botterweck, Ina Schaefer, and Timo Kehrer. 2019. Towards Efficient Analysis of Variation in Time and Space. In *23rd Intl. Systems and Software Product Line Conf. - Volume B*. ACM. https://doi.org/10.1145/3307630.3342414

[46] Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. 2018. A Classification of Product Sampling for Software Product Lines. In *22nd Intl. Systems and Software Product Line Conf. - Volume 1 (SPLC '18)*. Association for Computing Machinery. https://doi.org/10.1145/3233027.3233035

[47] Alan Wang, Nick Feng, and Marsha Chechik. 2023. Code-Level Functional Equivalence Checking of Annotative Software Product Lines. In *27th ACM Intl. Systems and Software Product Line Conf. - Volume A*. ACM. https://doi.org/10.1145/3579027.3608978

[48] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: how far are we?. In *35th IEEE/ACM Intl. Conf. on Automated Software Engineering*. ACM. https://doi.org/10.1145/3324884.3416590

[49] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. 2009. When private keys are public: Results from the 2008 Debian OpenSSL vulnerability. In *9th ACM SIGCOMM Conf. on Internet Measurement*.

[50] Bo Zhang and Martin Becker. 2012. Code-based variability model extraction for software product line improvement. In *16th Intl. Software Product Line Conf. - Volume 2*. ACM. https://doi.org/10.1145/2364412.2364428