TASTING: Reuse Test-case Execution by Global AST Hashing

Tobias Landsberg[1] [a], Christian Dietrich[2] [b] and Daniel Lohmann[1] [c]

[1]*Leibniz Universität Hannover, Germany*
[2]*Technische Universität Hamburg, Germany*

Keywords: Regression Test Selection, Testing, Continuous Integration, Static Analysis.

Abstract: We describe TASTING, an approach for efficiently selecting and reusing regression-test executions across program changes, branches, and variants in continuous integration settings. Instead of detecting changes between two variants of the software-under-test, TASTING recursively composes hashes of the defining elements with all their dependencies on AST-level at compile time into a *semantic fingerprint* of the test and its execution environment. This fingerprint is easy to store and remains stable across changes if the test's run-time behavior is not affected. Thereby, we can reuse test results across the history, multiple branches, and static compile-time variants. We applied TASTING to three open-source projects (Zephyr, OpenSSL, FFmpeg). Over their development history, we can omit between 10 percent (FFmpeg) and 95 percent (Zephyr) of all test executions at a moderate increase in build time. Furthermore, TASTING enables even higher savings across multiple checkouts (e.g., forks, branches, clones) and static software variants. Over the first changes to 131 OpenSSL forks, TASTING avoids 56 percent redundant test executions; for the Zephyr test matrix (64 variants), we reduce the number of test executions by 94 percent.

1 INTRODUCTION

Automated regression testing, that is, the repeated testing of an already tested program after a fine-grained software modification, has become standard practice (Yoo and Harman, 2012). However, testing takes considerable time and resources, so executing all tests after each change (the *retest-all* approach) is neither viable nor scalable (Rothermel et al., 1999). Hence, *regression-test selection (RTS)* (Rothermel and Harrold, 1996), which is the task of selecting a relevant test subset $T' \subseteq T$ for a given change $S \rightarrow S'$ to the *software-under-test (SUT) S*, remains a challenging problem. Such techniques are *sound* (sometimes called *safe* (Elbaum et al., 2014)) if they at least select those tests that reveal faults that were introduced by $S \rightarrow S'$.

In *continuous integration (CI)* settings, RTS becomes a lot more severe (Elbaum et al., 2014) as developers frequently merge their changes with the mainline (Duvall et al., 2007) and run ttest suites after each commit. Further, multiple branches and statically configured variants are often maintained in parallel. With the shift to decentralized version control systems,

branching development models did not only become ubiquitous, but the average size of commits also decreased by about 30 percent (Brindescu et al., 2014). With history rewriting, mailing-list patches, and patch-set evolution, the same (partial) change may even occur in different branches and commits (Ramsauer et al., 2019). As this can lead to thousands of to-be-retested versions per day (Elbaum et al., 2014; Memon et al., 2017), it becomes crucial to reuse test executions not only between two versions S and S' in a linear history but to track test results across *checkouts* (i.e., forks, branches, variants, and versions) of the SUT.

With conventional RTS, this would result in an $N \times N$-comparison matrix (N checkouts) as they work *change-based*, exactly comparing the two versions (S, S') to select the to-be-executed tests T'. While the granularity of this comparison differs (e.g., text-based (Vokolos and Frankl, 1997), statement (Rothermel and Harrold, 1996), function (Chen et al., 1994; Ren et al., 2004), file level (Gligoric et al., 2015)), the current version would have to be compared with a (potentially) large number of predecessors in a multi-checkout setting – which becomes too expensive with respect to computation time and disk space. Hence, change-based RTS approaches do not reuse test results across multiple checkouts but assume each as an independent linear history that needs separate testing.

[a] https://orcid.org/0000-0002-9792-7667
[b] https://orcid.org/0000-0001-9258-0513
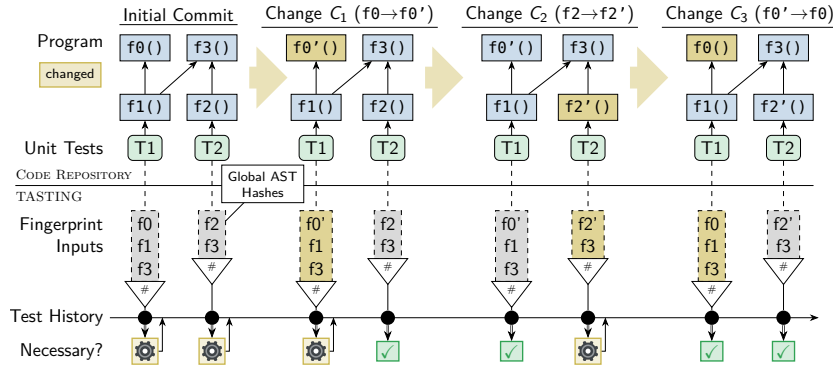[c] https://orcid.org/0000-0001-8224-4161

Figure 1: Application of TASTING to a history with four versions. For each test (T1, T2), we calculate the fingerprint from the AST hashes and consult the test history for fingerprint-result entries to avoid the test re-executions ($\checkmark$).

1.1 About this Paper

We propose TASTING, a new *content-based* and sound RTS strategy to efficiently reduce test re-execution across branches, histories, developers, and other sources of variation. Instead of comparing SUT variants for test selection, TASTING composes hashes of the software's defining syntactical elements (i.e., nodes in the *abstract syntax tree (AST)*) and their dependencies in a bottom-up manner. We integrate these hashes, in linear time, into a *semantic fingerprint* of the execution environment provided by the entry function of the concrete test. Fingerprints are guaranteed to alter for any change that might impact the test behavior – if they remain stable, we can avoid the re-execution. Thereby, test results can be efficiently stored and then reused over variants and an arbitrary complex branch or version history.

1.2 Our Contribution

For this paper, we claim the following contributions.

1. We present the concept of composable, hash-based *semantic fingerprints* that capture program behavior and enable arbitrarily-grained change impact analyses in linear time.

2. We present TASTING, an application of our approach to the RTS problem.

3. We evaluate TASTING with three open-source projects, where it omits up to 95 percent of all test executions at a moderate increase in build time.

In the following Sec. 2, we describe the TASTING approach in detail, followed by its implementation for RTS in Sec. 3. In Sec. 4, we evaluate and validate the approach and implementation. We discuss benefits, limitations, and threats to validity in Sec. 5, related work in Sec. 6, and finally conclude in Sec. 7.

2 SEMANTIC FINGERPRINTS

The TASTING approach is a method to characterize the potential run-time behavior of a test-case execution with a semantic fingerprint that is guaranteed to change if the behavior could change. By associating this over-approximating fingerprint with previously-run test executions, we can track results across multiple versions and checkouts. With a fingerprint–result database, we can reuse previous test results for new incoming changes. TASTING performs a static analysis within the compiler and in the linking stage to calculate and combine hashes over the AST.

Fig. 1 sketches our approach: In the build stage, we calculate a *global AST hash* for each function. In a reachability analysis from the test's entry function, we combine the hashes of all referenceable functions into the semantic fingerprint, which we use to search in the associative *test history*. For the *initial commit*, we execute all tests as the test history is still empty. The following C_1 and C_2 each modify a single function impacting *T1*'s or *T2*'s fingerprint, for which we re-execute the test case and store its result, while the other fingerprint is found in the test history and we omit re-execution. Although C_3 impacts *T1*, it reverts C_1, whereby the *T1*'s fingerprint changes back to its initial fingerprint.

2.1 System- and Test Model

The SUT consists of components (e.g., functions) that activate each other (i.e., call) to achieve the desired program behavior. Further, we allow for component references (i.e., function pointers) that are passed around and activated later on (i.e., indirect call) whereby we cover virtual functions and late dispatch. We demand that references are created explicitly and statically (i.e., obtained by taking a function's address but not by dy-

Local AST Hashing

Types — Abstract Syntax Tree

Global AST Hashing

Global Reference Graph — Global Hash Calculation

$$H(f) := \boxed{74} \leftarrow \boxed{74}$$
$$H(g) := \boxed{98} \leftarrow \boxed{62} \oplus H(f)$$
$$H(k) := \boxed{63} \leftarrow \boxed{15} \oplus H(f) \oplus H(GV)$$
$$H(GV) := \boxed{34} \leftarrow \boxed{34}$$

Recursive Group: m(), o()
$$H(m) := \boxed{c3} \leftarrow \boxed{21} \oplus \boxed{01} \oplus H(k)$$
$$H(o) := \boxed{27} \leftarrow \boxed{01} \oplus \boxed{21} \oplus H(k)$$

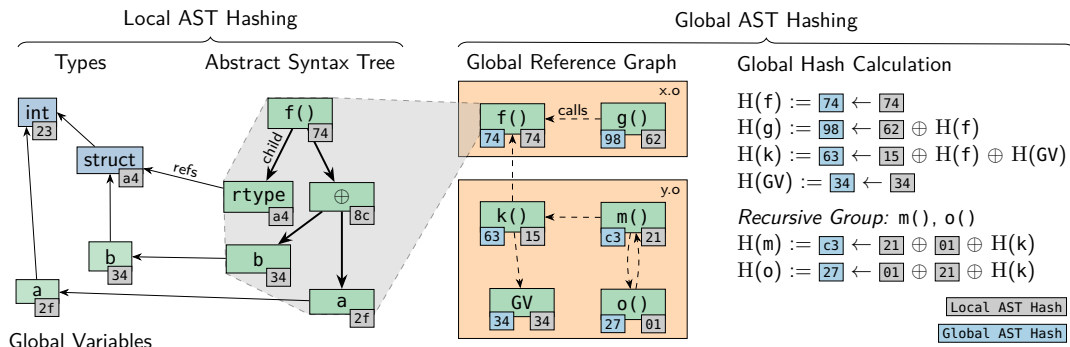Local AST Hash
Global AST Hash

Global Variables

Figure 2: Overview of the TASTING Approach for Global AST Hash Calculation.

namic introspection), that the source code is available, and that all activation sites and component accesses (to global variables) are statically known and extractable. Since we want to cover test-execution scenarios of statically-compiled languages, like C/C++, we consider these requirements as broadly applicable.

For each test, we require a list of the components-under-test, which can either be an explicit listing or a test program that adheres to the same rules as the SUT and that activates the to-be-tested components. We demand that tests are *stable*: For the same program version, a test either fails deterministically or it passes deterministically, but it never changes its result for different re-executions. If a test uses external files or network as inputs, they must be modeled as passive and stable components that the test accesses.

Without loss of generality, we will use a more concrete exemplary model for the rest of this paper: The SUT is a C program (or library) that is decomposed into different translation units, which are linked into a final binary. Each test is a separate program that we link with some (or all) of the SUT's translation units and that performs the functional testing by calling a subset of the SUT's functions. If a test depends on an external file, its contents get included as a global variable. In total, the test suite consists of separate and independent test binaries that validate different aspects of the SUT. This structure suits the timestamp-based change tracking of *make* and, hence, is common for many existing projects.

2.2 Local AST Hashes

We calculate a test's semantic fingerprint in two steps: First, we calculate the *local hash* for each function and each global variable, which captures the directly-enclosed syntactic AST nodes (e.g., initializers, statements) and the static cross-tree references (e.g., types, function declarations). In a second step (Sec. 2.3), we combine these local hashes into a *global hash* for each function. In combination, the global hashes of the

tested components make up the *semantic fingerprint* that identifies the version-specific test-case behavior.

For the first step, we employ *cHash* (Dietrich et al., 2017), which uses AST hashes to avoid unnecessary recompilations: cHash recursively visits (see Fig. 2) all AST nodes of a *translation unit (TU)* that influence the resulting binary and propagates hashes from the leaf nodes upwards in style of a Merkle tree (Merkle, 1982). For static cross-tree references (e.g., a variable definition references a type declaration), cHash calculates and includes the hash for the referenced node into the referrer-node hash. If the cross-tree reference points to a definition (e.g., has a function body), only the declaration (e.g., signature) is used. At the end, cHash compares the top-level hashes of the current and previous compilation run and aborts the compilation early on a match, avoiding costly optimization steps. While AST hashing is a technique to accelerate incremental rebuilds, it is also well-suited for fine-grained change-impact analyses.

In general, the AST hash of an object captures all elements (including type and global-variable declarations) that potentially influence the binary representation of that object; if the hash remains stable, the binary is guaranteed to remain stable. This property does not only hold true for the TU level but also for more fine-grained levels (i.e., function level) if we stop the upwards-propagation early. Furthermore, as cHash uses AST information, it is able to ignore purely textual changes, like coding-style updates or comment modifications. Therefore, we can use AST hashes of functions and global variables, which we will call *local hashes*, to identify equal variants across the history and different checkouts. Another benefit of cHash (which works as a Clang plugin) is its low overhead due to it operating on the AST, which is required for compilation anyhow, and its use of a fast non-cryptographic hash function.

While we will present TASTING as an extension of cHash, we are not limited to cHash but other fine-grained change impact analyses can be used as well.

For this, we demand that a component-local hash method $h()$ and a link function $l()$ which enumerates all directly referenced, accessed, or activated components, are available. For example, we could also calculate the function-local hashes over the compiler's intermediate representation or over the resulting assembler opcodes. Please note that we also treat passive elements, like global variables or virtual-function tables, as components. In Sec. 5, we will discuss the benefits of performing the static change-impact analysis on the AST level instead of IR or binary level.

2.3 Global Hash and Fingerprints

Local hashes have a static prediction quality: If two functions have the same local hash, they contain the same operations in the same order and structure. However, even if invoked with the same arguments, both can *behave* differently due to a call to a function with differing behavior. In order to lift the prediction from the static code level to the dynamic behavioral level, we calculate the *global AST hashes*, which we combine for each test case into the *semantic fingerprint*.

For this, we recursively define the global hash $H()$ with the local-hash function $h()$ and the link function $l()$, which spans a directed (potentially cycling) component-reference graph. In this graph, edges indicate a function call, an access to a global variable, or the calculation of a function- or global variable pointer. Thereby, global variables and their initialization values are modeled as leaf functions with no outgoing edges. On this graph, we also use a helper function $SC(f)$ that calculates the strongly connected subgraph for a function f.

$$H(f_0) = h(f_0) \oplus \underbrace{\left(\bigoplus_{f \in l(f_0) \setminus SC(f_0)} H(f) \right)}_{\textit{child functions}} \oplus \underbrace{\left(\bigoplus_{f \in SC(f_0) \setminus \{f_0\}} h(f) \right)}_{\textit{recursive group}}$$

The global hash of a function is the hashed concatenation ($\oplus$) of its own local hash and the global hashes of all its child functions. However, since many real-world programs contain recursion, we treat recursive graph-structures specifically: With $SC(f_0)$, we find all functions that are within the same recursive group as f_0 and could call f_0 recursively. For these, we include the *local* hashes to avoid cyclic dependencies for the hash calculation. In order to make the global-hash calculation as deterministic as possible, $SC()$ and $l()$ return function-name–sorted lists.

Fig. 2 shows a simplified example of global hash calculation: Since `f()` is a leaf function, its local hash (`74`) is directly used as the global hash (`74`). For `g()`, which calls `f()`, we combine its local hash (`62`) with

`f()`'s global hash. As function `k()` accesses the global variable `GV`, we include `f()`'s global hash as well as the hash of `GV` to account for its potential influence on `k()`'s behavior. To handle the strongly connected recursive group `{o(), m()}`, we only include their respective local hashes (`01`/`21`) into the global hash.

After the global-hash calculation, we derive the semantic fingerprint for a test case by collecting and hashing the global hashes of all relevant functions, which are provided by our test model (Sec. 2.1), for the respective test. Thereby, we cover all SUT functions that the test can call and all preparation code from the test case itself. In our exemplary model, it suffices to use the global hash of the test case's `main()` function as tests are self-contained executables.

Semantic fingerprints cover all potentially influencing functions, global variables, and initializers from the test and the SUT, and, therefore, they are an identifier for the test case's execution behavior. Rooted in the change-prediction quality of the local hash, test cases with an identical fingerprint will have the same test outcome. Thus, whenever a fingerprint appears for the second time on the same branch, a different branch, or even within a different source code repository, we can avoid re-execution and reuse the previous test result. Hence, it enables the creation of a fingerprint–result database in a large CI setup.

2.4 Soundness Considerations

In the following, we discuss that semantic fingerprints are a sound over-approximation to capture the influence of source code changes on the run-time behavior. Thereby, we assume that hash collisions are unproblematic. Otherwise, we could use a hash function (even cryptographic) with a smaller collision probability.

From our test model, we know that the same function, called with the same inputs in the same execution context (i.e., input parameters, global state), will yield the same result. Therefore, its behavior can change when (A) the function itself changes or (B) if its execution context changes.

For scenario A, we argue that a code change modifying the function's binary body will influence the function's local AST hash, which propagates to its global hash and the fingerprint. Consequently, as long as the local hash is sound, the global AST hash will also change on a scenario-A behavioral change.

In scenario B, we look at data that flows into the function: Every datum flowing into our function must be produced at some other point in the program. In our test model (see Sec. 2.1), where all inputs are expressed in the form of source code, data flows can only change if a source code change happened in another

```
typedef (bool)(fun_t*)();
fun_t selectFn() {            // global hash: d2 ↔ 5a
-   return &alwaysTrue;
+   return &alwaysFalse;
}
void exec(fun_t callback) {   // global hash: 73 ↔ 73
  if (callback())
    test_succeed();
  else
    test_fail();
}
void main() {                 // global hash: 6b ↔ 9c
  fun_t fn = selectFn();
  exec(fn);
}
```

Figure 3: Two programs (v1, v2) with an altered data flow.

part of the SUT. As long as the fingerprint covers those functions, we correctly capture the behavior of the test. It is important to note that a function's global hash can remain stable even if its input changes; only the combination of all relevant global hashes into the semantic fingerprint is predictive of the test's behavior.

To illustrate this, Fig. 3 shows a test that passes around a function pointer (with type `fun_t`), whose return value determines the result of the test-case execution. With the change v1→v2, `selectFn()` returns a different function pointer to `main()`, which feeds it into `exec()`. While this change changes the global hash of `selectFn()` and `main()`, the global hash of `exec()` remains stable, as `selectFn()` is *not* in its link set. Thus, in the context of the whole test, `exec()`'s hash remains stable although its behavior changes. Nevertheless, since TASTING will use the global hash of `main()`, which includes the hashes of the other two functions, as the test's fingerprint, it still correctly identifies the test-execution behavior.

3 IMPLEMENTATION

We base our TASTING prototype on the cHash (Dietrich et al., 2017) Clang plugin, which calculates local AST hashes (see Sec. 2.2) for C translation units. As cHash has only rudimentary support for C++, we are currently limited to C projects.

We modified cHash to export local hashes for each top-level function and each global variable of a TU. We inspect those AST nodes (i.e., calls and addressof) that create cross-component references, whereby we gather the necessary information for $h()$ and $l()$. We embed this information as a separate ELF (Executable and Linkable Format) section, which is discarded in the linking process, into the object files. As hash function, we use the non-cryptographic hash MurMur3[1].

Large projects use complex build systems, and often also custom linker scripts, to drive the static-

linking process. As it is crucial for us to know which components get included in a specific (test) binary, we instruct the linker to output its *cross-reference table (CRT)*, which describes which symbol got selected from which object file. In combination with the cHash-supplied data, we can build the complete reference graph $l()$ for the project. To also cover files without fine-grained data (e.g., compiled assembler), we hash the whole object file as a fallback. Since metadata generation is enabled via command-line switches and the cHash data is integrated as separate sections into the object files, TASTING is easy to integrate with complex build systems even if object files are collected and moved around (i.e., static libraries). From the developer's perspective, TASTING is as non-invasive as adding a few compiler- and linker flags.

For the global-hash calculation, we construct a *single* reference graph of all executables, test cases, and libraries, requiring the calculation of each global hash only once even if a function ends up in multiple executables. For each executable and test case (see Sec. 2.1), we use the global hash of the respective `main()` as the semantic fingerprint.

For the fingerprint–result database, we currently only store information for one previous build and compare the fingerprint of the to-be-executed test cases against that data set. However, a centralized fingerprint–result database that even works for a larger build–test farm could be built on the base of a simple key–value store, like `memcached`.

4 EVALUATION

For our evaluation, we use three open-source projects from different domains as case studies to validate our prototypical implementation and quantify its overheads and end-to-end time savings. For this, we apply TASTING to parts of the project's development history and compare build times, testing times, and the number of test-case executions. We also show that storability is the major benefit of semantic fingerprints by demonstrating the shortcomings of change-based RTS when it comes to a non-linear change history and static compile-time variants.

4.1 Case Studies

We apply TASTING to Zephyr[2], OpenSSL[3], and FFmpeg[4]. We chose these projects as they are open-source,

---

[1]https://github.com/aappleby/smhasher

[2]https://zephyrproject.org
[3]https://www.openssl.org
[4]https://ffmpeg.org

written in C, and are representative of different software classes (i.e., operating system, library, application). Also, they use different test-case execution schemes, requiring TASTING to be adaptable: Zephyr orchestrates its test suite with Python, OpenSSL utilizes Perl, and FFmpeg's test suite fully relies on `make`. From each source code repository, we selected 100-150 recent, successive commits and identified a set of relevant test cases.

Although we executed all relevant test cases, we differentiate between unit tests and integration tests. Because classifying a test based on its intention is difficult (Trautsch et al., 2020), we apply a technical definition: While integration tests execute binaries that are deployed to the user, unit-test binaries are purely built for testing external and internal APIs of the SUT. For example, in OpenSSL we label all tests that invoke the `openssl` binary as integration tests.

Zephyr is an embedded, scalable, real-time operating system that supports nine processor architectures and over 200 hardware boards. At the first evaluated commit, Zephyr had 18,263 KLOC[5], including 5.9 KLOC assembler, divided into 36 modules. Over all architectures and boards, the complete test suite comprises over 10,000 test cases. For our evaluation, we focus on the `native_posix` architecture and its regression-test suite of 398 tests, whereby we mimic the situation of a single CI job that runs the architecture-specific test suite for all incoming changes. Since Zephyr is built as a software product line, each regression test comes with its own *operating system (OS)* configuration and results in an application-specific OS library. Therefore, we do not differentiate between unit and integration tests, and we have to calculate one reference graph for each test case instead of sharing it between all tests for a given commit. Zephyr's repository has over 50,000 commits from which we selected the 150 latest ones (`73b29d68` to `5ee6793e`) for our evaluation.

OpenSSL is first and foremost a library that provides cryptographic primitives and TLS/SSL-secured connections, but it also ships the `openssl` tool that provides a UNIX-like interface for its cryptographic primitives. OpenSSL has 693 KLOC[5], including 76 KLOC assembler, divided into two libraries and 124 regression tests (including 96 unit tests). From the nearly 29,000 commits in the repository, we selected the 118 commits between the releases 1.1.1g and 1.1.1h, thereby covering a whole release cycle while staying in our target range of investigated commits. In contrast to Zephyr and FFmpeg, OpenSSL's test suite runs regression tests sequentially.
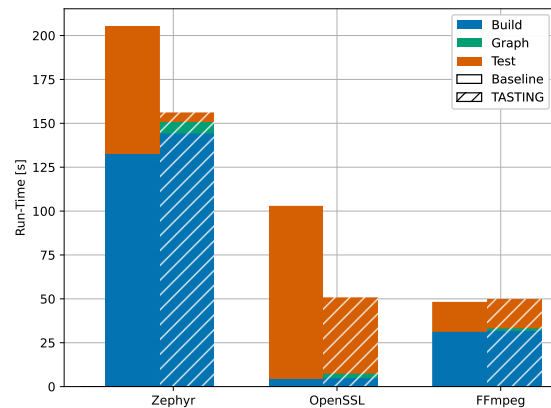

Figure 4: End-to-End Build and Test Times.

FFmpeg is a command-line application for audio and video processing. It had 1234 KLOC[5], including 101 KLOC assembler, divided into eight libraries and 3794 regression tests (including 344 unit tests). FFmpeg's repository has over 100,000 commits from which we used the 150 latest ones (`5e880774` to `f719f869`) for our evaluation.

4.2 Evaluation Setting

For each SUT, we used the unmodified source code and the default build configuration with a few minor adaptations for integrating TASTING into the build system. For example, we disabled configuration options (Zephyr: `BOOT_BANNER`) and excluded local hashes (OpenSSL: `OpenSSL_version()`) that carry uninterpreted commit and version information from our analysis. Further, we had to identify custom entry functions (Zephyr: `z_cstart()`) for the fingerprint calculation and modify FFmpeg's build system to separate build and test phases for our measurements.

Since TASTING targets a CI setting, we use a large server machine with two 24-core Intel® Xeon® Gold 6252 @ 2.10 GHz and 384 GB of memory for our evaluation. Because of SMT, 96 threads can actually run in parallel. As the software stack, we used Ubuntu 20.04 as the OS and Clang 10 as the compiler.

For each SUT, we iterate through the selected commit range and run a clean build, as it would be done by an CI setup, before running all tests. In this process, we measure the duration of the build phase, global-hash calculation, and test-suite execution separately. We can compare the end-to-end savings and the effectiveness of our approach by comparing it to the regular build process. Please note that the local-hash overheads are included in the build time, as local hashes are calculated by the modified cHash compiler plugin.

---

[5]Determined with `cloc` on the SUT's repository
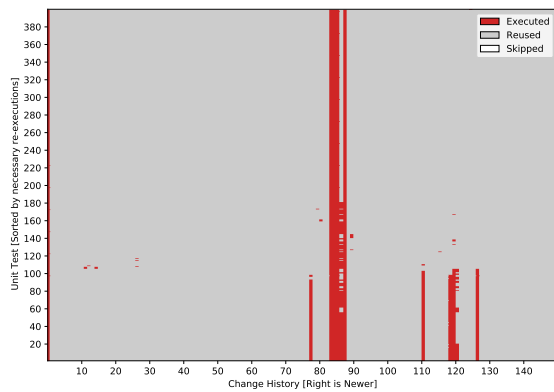
(a) Zephyr
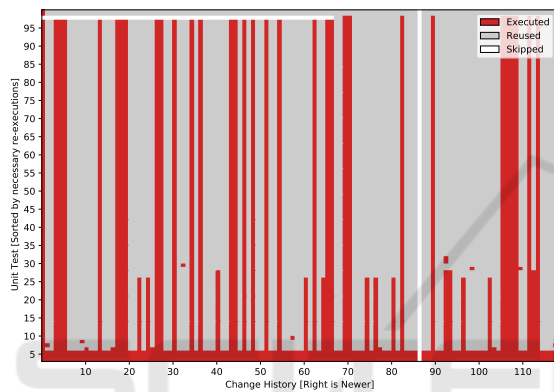
(b) OpenSSL
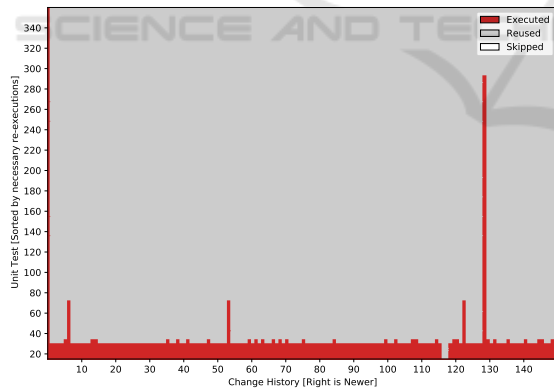
(c) FFmpeg

Figure 5: Unit-Test Execution Matrix. For the selected test *subset* and over the investigated change history, we show unit-tests that have to be executed (red) and executions for which we could reuse a previous test result (gray). Previously missing as well as skipped tests are white.

4.3 Validation of TASTING

To validate TASTING, we compare our RTS set for a given commit with a behavior-change detection: During the test-case execution, we dynamically trace all

function calls with the Valgrind tool[6] and extract the respective function bodies from the binary. Thereby, we are able to compare different test-case behaviors by comparing their respective binaries filtered by the called functions. If the set of called functions changes or if the selective binary comparison indicates a change, we assume that the behavior of the test case actually changed. For a successful validation, TASTING must schedule a test for re-execution whenever the behavior of the executed test changed.

We chose this validation method to demonstrate TASTING's ability to handle function pointers, which can introduce dynamic behavior – a problem area for static RTS methods. The dynamic-tracing approach is based on the assumption that a test case's run-time behavior is uniquely identifiable by the instructions of the executed function. Thereby, this approach is stricter than TASTING since it only considers functions that are actually called instead of all potentially called or referenced functions.

For the validation, we chose OpenSSL since it has a high proportion of unit tests (unlike FFmpeg), shows a high rate of changed test-case behavior (unlike Zephyr) and utilizes function pointers. Since TASTING is most effective for unit tests, we solely focused on the 117 OpenSSL unit-test cases. For each investigated commit, we compare the set of tests with a changed dynamic behavior with the set of re-executions that was scheduled by TASTING. Whenever a behavioral change was detected, we demand that the semantic fingerprint must also differ from the previous commit. In all cases, TASTING predicted all actually observed behavioral changes correctly.

We also compared the number of transitively included local hashes with the number of actually called functions. While the average unit test calls 923 functions, TASTING includes 3904 local hashes into a fingerprint on average, which resembles 40 percent of all functions.

4.4 End-to-End Costs and Savings

As we aim for CI settings, we simulated this workload by applying each evaluated change for the respective SUT and performing a parallelized, clean build, and running the test suite with and without TASTING.

Overheads. During the build step, the calculation of local hashes and the generation of the CRT introduces overheads (see Fig. 4). The build time increased by 8.9 percent (11.8 s) for Zephyr and 14 percent (0.65 s) for OpenSSL. For FFmpeg the mean build time in-

---

[6]https://valgrind.org

creased by only 1.1 percent (0.36 s) because the number of assembler files, for which we did not introduce any overhead, is considerably larger compared to the other projects. While TASTING introduces a measurable overhead into the build process, we could use the full cHash approach and abort redundant compilations early, potentially hiding TASTING's overheads by cHash's savings.

In addition to longer build times, TASTING needs to create the reference graph and calculate the semantic fingerprints before the tests can be executed. On average, this took 2.6 seconds for OpenSSL, 1.6 seconds for FFmpeg, and 6.3 seconds for Zephyr. The longer times for Zephyr stem from the necessity to construct one reference graph for each test case instead of using the same graph for all tests of the same commit.

Savings. Using the semantic fingerprints, TASTING prevents redundant regression-test executions with the goal to improve test times. In summary, over all test-case executions, we could avoid 95 percent for Zephyr, 66 percent for OpenSSL, and 10 percent for FFmpeg. Overall, this resulted (see Fig. 4) in an average end-to-end reduction of the build-and-test time by 24 percent for Zephyr and 50 percent for OpenSSL. However, for FFmpeg, TASTING increased the mean time spent on a single commit by 2 percent. In Fig. 5, we show the prevented and the necessary *unit-test* executions, leaving aside the integration tests, for the investigated commit ranges. Integration tests do not fit our test model (Sec. 2.1) well, which we will discuss in Sec. 5.3.

The increased end-to-end time for FFmpeg has two reasons: First, FFmpeg makes heavy use of manually-implemented dynamic dispatch, which results in a gap between the static reference graph and the actual activation patterns: In C programs, dynamic dispatch is usually implemented using a struct containing a set of function pointers. If any function in such a set changes its local hash, all functions (or unit-tests) referencing that struct end up with a different global hash. We will discuss this topic further in Sec. 5.4.

Second, while we prevent 94 of FFmpeg's unit-test executions (see Fig. 5c), only 9 percent of FFmpeg's test cases are unit tests. While, on average, 94 percent of the unit-test executions were avoided, we could only avoid 2 percent of the integration tests. While this reduced the overall test time by 0.89 seconds (5.1 percent), the time for calculating fingerprints (+1.93 s) outweighed the achieved savings in the test execution.

For Zephyr (Fig. 5a), we see that after the first commit, where all test cases had to be executed, TASTING only had to execute a few test cases for each change. Only between changes 80 and 90, where the

basic kernel primitives, which are used in all test cases, were refactored, we see an increased need for test executions. These good results stem from the fact that changes in Zephyr are usually very small and only touch a single subsystem. For example, out of the 150 evaluated commits, most changes occurred in the submodules drivers (24), documentation (18), network (16), and Bluetooth (14)[7], meaning most of the other subsystems (and regression tests) are usually unaffected. Furthermore, changes to other architectures (e.g., ARM) do not affect the test suite of our focus architecture (`native_posix`), which TASTING successfully exploited. It is likely that the other architectures supported by Zephyr show a similar reduction pattern, which would result in additional end-to-end savings if compared to the retest-all approach applied to all architectures.

For OpenSSL, the unit-test matrix in Fig. 5b shows a different test-execution pattern: It is noticeable that a few unit tests were run for every change. These tests run scripts instead of a binary, which is outside TASTING's scope, and we re-execute them if in doubt. It is also noticeable that either these few tests were executed, or around 25 percent, or the complete unit-test suite. This could indicate that OpenSSL uses a coarse-grained unit test suite or it might be due to OpenSSL using manually-implemented dynamic dispatch. Please note, that the horizontal white line is a later introduced test case and the vertical white line is a commit that did not compile.

4.5 Cross-checkout Savings

Up to this point, we only looked at the change history of a *single* source code repository and how we can reuse test executions over this history. While TASTING's fingerprint calculation is more efficient by design, change-based approaches could, in principle, avoid the same number of tests if given a linear history. However, with the modern decentralized-development paradigm, multiple *checkouts* with diverging histories exist in branches, forks, or as local clones on a CI bot or a (poorly connected) developer machine. In such a setting, change-based RTS approaches typically need to assume an independent linear history for each checkout and, hence, also have to re-execute all tests per checkout. To avoid this, it would be necessary to (a) store all the intermediate data required to detect a change together with each and every commit, as every commit could be the base (previous) of a branch, fork, or local clone, and (b) compare each checkout against

---

[7]We automatically extracted subsystem tags for each change from developer annotations in the respective commit messages.

a potentially large number of predecessors. This would require a lot of disk space and computation time.

With our content-based fingerprints, we can cut down on these problems, considering fingerprints require only minimal storage space and are very fast to compare. For example, when storing 128 bit hashes and a 1-bit test outcome (i.e., passed, failed), we need less than 2 kiB for all hashes per one OpenSSL version (e.g., a commit) with its 124 test cases. Even better, if we use a central *fingerprint–test-result store* (for instance, a memcached server), only newly found fingerprints have to be saved. In case of OpenSSL, where 66 percent of tests executions were avoided, we would have to store results for 5304 test-case executions (83.52 kiB) for the analyzed 118 commits.

To give a more comprehensive view on the cross-checkout savings, we systematically analyzed all 2463 forks of OpenSSL created in 2019 and 2020 on GitHub. From these, 183 had actual changes (at least one commit ahead of mainline) and 131 of them compiled without error. We compared the required test executions after the first change, which mimics the workflow of a developer that checks out a repository, makes a change, and runs the test suite. While a change-based RTS would re-execute all tests, TASTING with a global fingerprint store avoids 56 percent of all test executions because their fingerprints were already known from the original OpenSSL repository. As stored fingerprints are independent of the current checkout, developers and CI bots can avoid test-case executions whenever they clone a repository or switch between branches.

4.6 Cross-variant Savings

Another strength of the content-based RTS strategy of TASTING is that it also trivially covers test execution across static compile-time variants: Configurable software, like Linux or Zephyr, provides thousands of variants determined at compile time by means of conditional compilation (i.e., #ifdef blocks). A change could potentially affect a large number of variants. In this setting, ensuring just successful builds is already an enormous task (Tartler et al., 2014; Kerrisk, 2012). Running regression tests on each variant build afterward is even more resource-intensive.

Again, to solve this, a change-based RTS would require a known previous state to compare against. This could be either (a) the state of this variant before the change in a linear history or (b) another variant from the same version. (a) would require to store test selection data for *all* variants with each commit, which would take considerable disk space. For (b), it would be necessary to select a specific variant to compare against; finding the best variant sequence (i.e., with

the highest number of omitted tests) is a combinatorial problem.

A central fingerprint store circumvents all of these problems because it inherently covers checkouts and variants. To demonstrate this, we ran the tests for multiple variants in a checkout of Zephyr, which mimics the typical developer tasks of testing all customer-specific variants before shipping a new version. In the Zephyr configuration system, we simply picked the first six features that actually impact the test suite (many drivers are not covered by Zephyr's POSIX tests) for permutation,[8] resulting in 64 variants.

A change-based RTS system could have avoided between 24 and 62 percent of test executions over all variants, depending on variant comparison sequence.[9]

TASTING with a central fingerprint store, however, is inherently sequence-agnostic (and, thus, also trivially parallelizable) and could avoid 94 percent of test executions. After 23 variants, 95 percent of the actually required test executions were already completed, which further demonstrates the effectiveness of a fingerprint store.

5 DISCUSSION

With TASTING, we propose a *content-based* strategy to avoid unnecessary test-case executions by identifying and reusing test-case executions from previous test-suite runs. In contrast to change-based strategies that identify the differences between two program versions, we see three major benefits of our content-based approach: complexity reduction, storability of results, and language interoperability.

5.1 Benefits of Our Approach

First, TASTING's static analysis has linear complexity with regard to the program size as no fine-grained matching between program versions is necessary: For the local hashing, we visit each AST node exactly once and propagate hashes from the bottom to the top. Furthermore, if we leave aside recursive groups, we incorporate every local hash exactly once into a global hash, which is otherwise only dependent on its local environment in the reference graph. Thereby, TASTING is able to keep its overheads moderate, which allows us to actually harvest test-case avoidances as end-to-end savings in projects with a fine-grained unit-test suite.

---

[8]namely `ASSERT`, `BT`, `CBPRINTF_COMPLETE`, `LOG`, `DEBUG`, `THREAD_STACK_INFO`

[9]Calculated by random sampling (48 million samples) as 64 variants lead to $1.3 \cdot 10^{89}$ possible testing sequences.

On a higher level, the storability of test-case fingerprints and global hashes allows for test-avoidance strategies that are harder to achieve with change-based strategies: As storing, finding, and comparing fingerprints is fast, reusing test executions across branches, repositories, and variants – and any combination thereof – does not require a quadratically-growing comparison matrix between N program versions. This reduced complexity harvests the insight that it is not necessary to calculate the actual difference between two programs to avoid the test execution but that it is sufficient to know a fingerprint that identifies the complete test behavior. Moreover, as the calculation of semantic fingerprints is abstracted by the local-hash function and the link function, the TASTING approach promises interoperability between different programming languages. Since the local-hash function encapsulates the language-specific change summary, the reference graph, which has a broad understanding of "references" (i.e., address calculation, access, activation), can remain language-agnostic. Even in cases where no language-specific hash function is available, we can fall back to hashes of the build artifacts. We used this technique (see Sec. 3) to incorporate assembler files into our RTS approach.

5.2 Level of Local-hash Calculation

Another aspect to discuss is our decision to perform our static analysis on the AST level instead of other abstraction levels within the compilation process (source code $\rightarrow$ AST $\rightarrow$ *immediate representation (IR)* $\rightarrow$ binary). While local hashing is possible on all mentioned levels, the AST level has some benefits: As the program structure only becomes visible after parsing and the semantic analysis, a fine-grained dependency analysis between program elements is only possible from the AST level downwards. At the other end, component references, especially addressof-calculations and data accesses, are hard to spot on the binary level as they become indistinguishable from other immediate operands. While AST and IR-level are semantically quite close to each other, AST-level hashing has a higher potential to shadow its own computational overheads by aborting the compilation earlier. On the other hand, the IR level (e.g., LLVM IR) is often designed to be language-agnostic, which makes supporting different programming languages easier. Another aspect is the closeness of the AST level to the programmer's intention, while the IR level is closer to the final binary. For example, it might surprise the developer who added a `const` keyword that some tests are not executed because there was no impact on the IR code. Hence, AST-level hashing, although more sensitive to

changes than an IR-level analysis, follows the principle of the least surprise more closely. However, in the end, the TASTING approach is applicable, even in a combined fashion, for different programming languages on the AST- and the IR level.

5.3 Static vs. Dynamic RTS

TASTING is a purely static approach to RTS that only uses the program's control structure (i.e., call hierarchy). We over-approximate interprocedural data flows (i.e., function pointers) by incorporating the global hashes of all functions that *could* act as a source in order to calculate a safe test-case fingerprint. While we thereby avoid costly interprocedural data-flow analyses, this comes at the cost of re-running test cases more often than necessary. We can quantify this over-approximation if we compare the number of actually called functions with the number of functions whose local hash influences a semantic fingerprint. For the average OpenSSL test case, 40 percent of all functions influence the semantic fingerprint, while only 9 percent are actually called during the test-case execution.

Therefore, it would be beneficial to combine our approach with dynamic tracing to narrow down the link function: For example, it should be possible to reuse a dynamically-traced reference set (e.g., from a previous execution) instead of the statically-deduced one as long as all local hashes in the call hierarchy from the program entry down to a given function are equal. Thereby, we know that no additional outgoing edge can appear and the fingerprint calculation remains sound. We consider this a promising topic for further research.

5.4 Threats to Validity

In the following, we discuss potential threats to the validity of our results and the generalizability of our approach to other languages and program structures.

With our case studies (Zephyr, OpenSSL, and FFmpeg), we have selected projects that use C as their programming language and that come with a reasonably sized test suite. We chose these projects as being representatives of different classes of programs (embedded operating system, library, command-line tool) and testing strategies (unit tests vs. integration tests) that show the benefits with fine-grained unit tests (OpenSSL) as well as the limitations regarding coarse-grained integration tests (FFmpeg).

With our validation, we could show that our approach never missed a changed function. However, as we have not compared actual execution traces, but the opcodes of all called functions, there is the chance

that our validation has missed a behavioral change that was also missed by TASTING. Nevertheless, given that Valgrind recorded all calls correctly, such a miss could only stem from a change that only touched the initial values of a global variable. Since TASTING explicitly includes the local hashes of such variables into those functions that access or reference them, we are confident that TASTING would behave correctly even if the validation missed the change.

For the applicability of our approach, we see the necessity to enumerate all data sources for each test case as a major challenge for the integration into existing test systems. Normally, these dependencies are not as explicit as required by TASTING. However, with file-grained tracing methods, like EKSTAZI (Gligoric et al., 2015), such dependencies can be discovered and integrated on a coarse-grained level. Similarly, we can integrate components that are written in languages without local-hashing support on the file-grained level, as we have done for assembler source code.

The largest obstacle for the generalizability of our approach for other programming languages is the link function. For our static analysis, we assume that the statically-derivable reference graph is largely equal to the dynamically-observable references. However, if this over-approximation is too imprecise for a given language, it can cause every function to influence every test-case fingerprint, resulting in no end-to-end savings. For example, for a scripting-language interpreter (e.g., Python), the actual call-hierarchy is largely driven by the interpreted program – not by the static structure of the interpreter loop. In such cases, our approach would work better on the level of the interpreted language. An alternative would be a combination of TASTING with fine-grained function tracing.

6 RELATED WORK

Regression testing, and, in our case, more specifically RTS, is a topic that has attracted a lot of attention in the last 30 years, as surveyed in several large literature reviews (Biswas et al., 2011; Engström et al., 2010; Yoo and Harman, 2012). Because of the large body of research, we will only give a short roundup of important RTS techniques before we discuss other content-based caching techniques that inspired TASTING.

Regression-test Selection. Many RTS techniques use a two-step approach: (1) For each test case, they derive the set of covered program entities that is used or validated by the given test. (2) They compare two versions and derive a set of changed program entities and intersect it with each test's dependencies to select

or dismiss it for re-execution. From these methods, TASTING differs fundamentally since we do not compare two versions but derive a semantic fingerprint from a single version and associate it with the test result.

One dimension RTS techniques differ in is the *granularity* of entities that is used for the test-dependency detection. There are techniques that work on the textual level (Vokolos and Frankl, 1997), on the data-flow level (Harrold and Souffa, 1988; Taha et al., 1989), on the statement level (Rothermel and Harrold, 1996), on the function level (Chen et al., 1994), on the method level (Ren et al., 2004), on the class level (Orso et al., 2004), on the module level (Leung and White, 1990), on the file level (Gligoric et al., 2015), or on the level of whole software projects (Elbaum et al., 2014; Gupta et al., 2011). With HyRTS (Zhang, 2018), a method that uses a varying granularity depending on the change is also available. In general, it was noted (Gligoric et al., 2015) that a finer granularity results in higher analysis overheads but also provides less severe over-approximations. For TASTING, we choose the function-level granularity, because calling functions is the technical link between test case and SUT. However, as local-hash calculation works on the AST, which captures the hierarchical organization of program entities, other granularities are also possible.

Another dimension is the method to detect dependencies between program entities. This can either be achieved completely statically (Kung et al., 1995; Ren et al., 2004; Rothermel and Harrold, 1996) or by inspecting recorded test-case–execution traces (Gligoric et al., 2015; Orso et al., 2004; Chen et al., 1994). While it is easier to argue the soundness of the static methods, dynamic methods result in smaller dependency sets, which reduces the frequency of unnecessary re-executions. In this dimension, TASTING uses a purely static analysis method to calculate its link function, but a combination with dynamic trace information should be possible without compromising on soundness.

Most similar to TASTING is EKSTAZI (Gligoric et al., 2015), which works on the file level and dynamically traces files that a given test accesses (e.g., Java `.class` files). For these files, it calculates a content-based hash and executes those test cases whose accessed files changed. While they provide a *smart-hashing* method that hides unnecessary information (e.g., build dates) from the hash function, they only use those hashes to identify changes on the file level, making it a change-based RTS method. TASTING not only uses a more fine-grained method to include only relevant information into the hash but also uses the content-based hash to identify test-execution results.

In CI environments, the test load on the CI server is reduced by enforcing some pre-submit testing on the developer's local machine, so that developers get feedback quickly and fewer tests fail eventually on the server (Elbaum et al., 2014). With TASTING it would be possible to also reuse local test executions across the whole organization, as local test execution results could easily be submitted together with the changes.

Content-based Incremental Compilation. Most inspiring for TASTING were recent advancements in incremental compilation techniques that replace the decades-old approach of timestamp-based `make` (Feldman, 1979) with a content-based paradigm. These content-based methods for incremental compilation summarize the input data with the use of a hash function and compare it to the previous build or manage some kind of hash–build-artifact database. For example, the `ccache`[10] tool hashes the preprocessed C/C++ compiler input and manages a cache of object files. Also, both Microsoft and Google (York, 2011; Esfahani et al., 2016) use a similar textual hashing to access a distributed object-file cache. As described in Sec. 2.2, the cHash method achieves an even higher cache hit rate by using the parsed program instead of its textual notation. Similar to the RTS problem, not only static methods but also dynamic dependency-detection mechanisms are available: For example, Memoize (Mc-Closkey, 2007) and Fabricate (Technology,), which were inspiring for Gligoric et al.(Gligoric et al., 2015), use the Linux tool `strace` to record all accessed files and calculate an MD5 hash with the goal of avoiding unnecessary build steps. With TASTING, we provide a method that lifts the content-based recompilation avoidance from the build step to the testing stage.

7 CONCLUSIONS

We presented TASTING, a content-based fingerprinting technique for identifying the behavior of deterministic programs that builds upon AST hashing. TASTING statically summarizes all defining elements of the compiled program into a *semantic fingerprint*, using a standard hash function. Whenever the behavior of the program changes, the fingerprint will also differ. Thereby, TASTING provides an efficient implementation of the *regression-test selection (RTS)* problem, where the results of executed tests could easily be stored and later be reused by their unique fingerprint across versions, branches, repositories, and variants.

---

[10]https://ccache.dev

In our evaluation with Zephyr, OpenSSL, and FFmpeg, we could avoid 95 percent of all test executions for Zephyr, 66 percent for OpenSSL, and 10 percent for FFmpeg in a CI setting with sequentially applied commits to the master branch. Since in the modern decentralized-development paradigm change histories often diverge (e.g., branches, forks, local clones), we also showed that we can avoid 56 percent of all test executions by reusing results across histories on the basis of 131 publicly-available OpenSSL forks. Testing configurable software also benefits from our approach, shown for 64 variants of Zephyr, where we could avoid 94 percent of all test executions.

ACKNOWLEDGEMENTS

REFERENCES

Biswas, S., Mall, R., Satpathy, M., and Sukumaran, S. (2011). Regression test selection techniques: A survey. *Informatica*, 35(3).

Brindescu, C., Codoban, M., Shmarkatiuk, S., and Dig, D. (2014). How do centralized and distributed version control systems impact software changes? In *36th Intl. Conf. o. Software Engineering*, ICSE 2014, New York, NY, USA. Association for Computing Machinery.

Chen, Y.-F., Rosenblum, D. S., and Vo, K.-P. (1994). Testtube: A system for selective regression testing. In *16th Intl. Conf. o. Software Engineering*.

Dietrich, C., Rothberg, V., Füracker, L., Ziegler, A., and Lohmann, D. (2017). cHash: detection of redundant compilations via AST hashing. In *2017 USENIX Annual Technical Conference*, Berkeley, CA, USA. USENIX Association.

Duvall, P. M., Matyas, S., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley.

Elbaum, S., Rothermel, G., and Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *22nd ACM SIGSOFT Foundations of Software Engineering*.

Engström, E., Runeson, P., and Skoglund, M. (2010). A systematic review on regression test selection techniques. *Information and Software Technology*, 52(1).

Esfahani, H., Fietz, J., Ke, Q., Kolomiets, A., Lan, E., Mavrinac, E., Schulte, W., Sanches, N., and Kandula, S. (2016). Cloudbuild: Microsoft's distributed and caching build service. In *38th Intl. Conf. o. Software Engineering Companion*.

Feldman, S. I. (1979). Make — a program for maintaining computer programs. *Software: Practice and experience*, 9(4).

Gligoric, M., Eloussi, L., and Marinov, D. (2015). Practical regression test selection with dynamic file dependencies. In *2015 Software Testing and Analysis*.

Gupta, P., Ivey, M., and Penix, J. (2011). Testing at the speed and scale of google.

Harrold, M. J. and Souffa, M. (1988). An incremental approach to unit testing during maintenance. In *1988 Conference on Software Maintenance*.

Kerrisk, M. (2012). Kernel build/boot testing. https://lwn.net/Articles/514278/, accessed 28. Feb 2022.

Kung, D. C., Gao, J., Hsia, P., Lin, J., and Toyoshima, Y. (1995). Class firewall, test order, and regression testing of object-oriented programs. *JOOP*, 8(2).

Leung, H. K. and White, L. (1990). A study of integration testing and software regression at the integration level. In *Conference on Software Maintenance 1990*.

McCloskey, B. (2007). Memoize. https://github.com/kgaughan/memoize.py, accessed 28. Feb 2022.

Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., and Micco, J. (2017). Taming google-scale continuous testing. In *39th Software Engineering: Software Engineering in Practice Track*.

Merkle, R. C. (1982). Method of providing digital signatures. US Patent 4,309,569.

Orso, A., Shi, N., and Harrold, M. J. (2004). Scaling regression testing to large software systems. *ACM SIGSOFT Software Engineering Notes*, 29(6).

Ramsauer, R., Lohmann, D., and Mauerer, W. (2019). The list is the process: Reliable pre-integration tracking of commits on mailing lists. In *41st International Conference on Software Engineering*.

Ren, X., Shah, F., Tip, F., Ryder, B. G., and Chesley, O. (2004). Chianti: a tool for change impact analysis of java programs. In *OOPSLA'04*.

Rothermel, G. and Harrold, M. J. (1996). Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8).

Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (1999). Test case prioritization: An empirical study. In *IEEE Software Maintenance*, USA.

Taha, A.-B., Thebaut, S. M., and Liu, S.-S. (1989). An approach to software fault localization and revalidation based on incremental data flow analysis. In *13th Intl. Computer Software & Applications Conf.*

Tartler, R., Dietrich, C., Sincero, J., Schröder-Preikschat, W., and Lohmann, D. (2014). Static analysis of variability in system software: The 90,000 #ifdefs issue. In *2014 USENIX Annual Technical Conference*, Berkeley, CA, USA. USENIX Association.

Technology, B. Fabricate. https://github.com/brushtechnology/fabricate accessed 28. Feb 2022.

Trautsch, F., Herbold, S., and Grabowski, J. (2020). Are unit and integration test definitions still valid for modern java projects? an empirical study on open-source projects. *Journal of Systems and Software*, 159.

Vokolos, F. I. and Frankl, P. G. (1997). Pythia: A regression test selection tool based on textual differencing. In *Reliability, quality and safety of software-intensive systems*. Springer.

Yoo, S. and Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2).

York, N. (2011). Build in the cloud: Distributing build steps. http://google-engtools.blogspot.de/2011/09/build-in-cloud-distributing-build-steps.html, accessed 7. Feb 2017. [Online; posted 23-09-2011].

Zhang, L. (2018). Hybrid regression test selection. In *40th Intl. Conf. o. Software Engineering*.