Sandiff: Semantic File Comparator for Continuous Testing of Android Builds

Carlos E. S. Aguiar, Jose B. V. Filho, Agnaldo O. P. Junior,
Rodrigo J. B. Fernandes, Cícero A. L. Pahins
Sidia: Institute of Science and Technology
Manaus, Brazil
Emails: {c.eduardo, jose.vf, agnaldo.j, rodrigo.f, cicero.p}@samsung.com

***Abstract*—With ever-larger software development systems consuming more time to perform testing routines, it is necessary to think about approaches that accelerate continuous testing of those systems. This work aims to allow the correlation of semantic modifications with specific *test cases* of complex suites, and based on that correlation, skip time-consuming routines or mount lists of priority routines (*fail-fast*) to improve the productivity of mobile developers and time-sensitive project deliveries and validation. In order to facilitate continuous testing of large projects, we propose Sandiff, a tool that efficiently analyzes semantic modifications of files that impacts domain-specific testing routines of the official Android Test Suite. We validate our approach on a set of real-world and *commercially-available* Android images of a large company that comprises two major versions of the system.**

Keywords–Testing; Validation; Continuous; Tool.

I. Introduction

As software projects get larger, continuous testing becomes critical, but at the same time, difficult and time-consuming. Consider a project with a million files and intermediate artifacts. It is essential that a *test suite* that offers *continuous testing* functionalities performs without creating bottlenecks or impacting project deliveries. However, effectively using continuous integration can be a problem: tests are time-consuming to execute, and by consequence, it is impractical to run complete modules of testing on each build. In these scenarios, it is common that teams lack *time-sensitive* feedback about their code and compromise user experience.

The testing of large software projects is typically bounded to robust test suites. Moreover, the quality of testing and validation of ubiquitous software can directly impact people's life, a company's perceived image, and the relation with its clients. Companies inserted in the Global Software Development (GSD) environment, i.e., with a vast amount of developers cooperating across different regions of the world, tend to design a tedious process of testing and validation that becomes highly time-consuming and impacts the productivity of developers. Moreover, continuous testing is a *de facto* standard in the software industry. During the planning of large projects, it is common to allocate some portion of the development period to design testing routines. Test-Driven Development (TDD) is a well-known process that promotes testing before feature development. Typically, systematic software testing approaches lead to compute and time-intensive tasks.

Sandiff is a tool that helps to reduce the time spent on testing of large Android projects by enabling to skip domain-specific routines based on the comparison of meaningful data without affecting the functionality of the target software. For instance, when comparing two Android Open Source Project (AOSP) builds that were generated in different moments, but with the same source code, build environment and build instructions, the final result is different in byte level (byte-to-byte), but can be semantically equivalent based on its context (meaning). In this case, it is expected that these two builds perform the same. However, the problem is proving it. Our solution relies on how to compare and prove that two AOSP builds are semantically equivalents. Another motivation is the relevance of Sandiff to the continuous testing area, where it can be used to reduce the time to execute the official Android Test Suite (VTS). As our solution provides a list of semantically equivalent files, it is possible to skip tests that validate the behavior provided by these files. Consider the example of Figure 1 in which the official Android Test Suite was executed in a *commercially-available* build based on AOSP. The execution of all modules exceeded 4 hours, compromising developer performance and deliveries.

Suite/Plan	VTS/VTS
Suite/Build	9.0_R9 / 5512091
Host Info	seltest-66 (Linux - 4.15.0-51-generic)
Start Time	Tue Jun 25 16:17:23 AMT 2019
End Time	Tue Jun 25 20:39:46 AMT 2019
Tests Passed	9486
Tests Failed	633
Modules Done	214
Modules Total	214
Security Patch	2019-06-01
Release (SDK)	9 (28)
ABIs	arm64-v8a,armeabi-v7a,armeabi

Figure 1. Summary of the official Android Test Suite – *Vendor Test Suite* (VTS) – of a *commercially-available* AOSP build.

By *comparison of meaningful data*, we mean comparison of sensitive regions of critical files within large software: different from a byte-to-byte comparison, a semantic comparison can identify domain-related changes, i.e., it compares sensitive code paths or *key-value* attributes that can be related to the output of a large software. By *large*, we mean software designed by a vast amount of developers that are inserted in a distributed software development environment whereupon automatic test suits are necessary.

In summary, we present the key research contributions of Sandiff:

- (i) An approach to perform *semantic* comparison and facilitate *continuous testing* of large software projects.
- (ii) An evaluation of the impact of using Sandiff in real-world and *commercially-available* AOSP builds.

Our paper is organized as follows. In Section II, we provide an overview of *binary* comparators and their impact

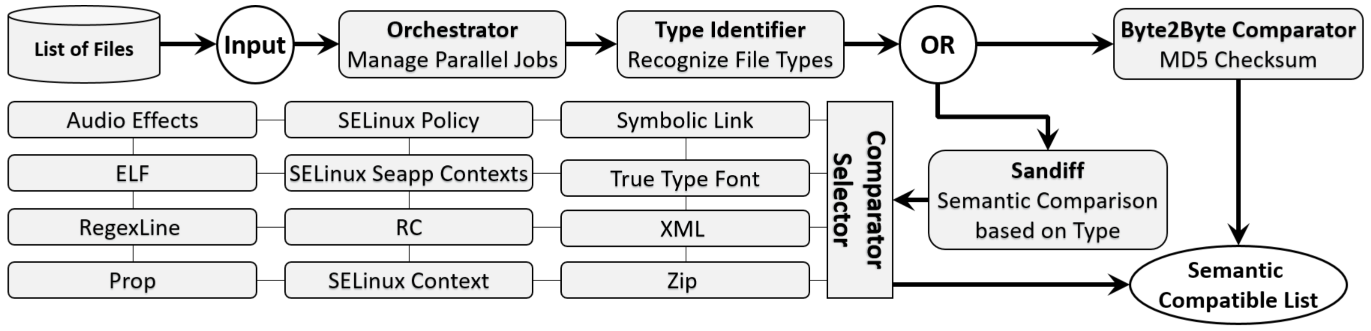

Figure 2. Sandiff verifies the semantic compatibility of two files or directories (list of files) and report their differences.

on continuous testing of large projects. In Section III, we describe Sandiff and its main functionalities: (i) input detection, (ii) content recognition, and (iii) specialized semantic comparison. In Section IV, we present the evaluation of Sandiff in *commercially-available* builds based on AOSP and discuss the impact of continuous testing of those builds. We conclude the paper with avenues for future work in Section V.

II. Related Work

To the best of our knowledge, few approaches in the literature propose *comparison* of files with different formats and types. Most of the comparison tools focus on the comparison based on diff (text or, at most, byte position). Araxis [1] is a well-known commercial tool that performs three types of file comparison: text files, image files, and binary files. For image files, the comparison shows the pixels that have been modified. For binary files, the comparison is performed by identifying the differences in a byte level. Diff-based tools, such as Gnu Diff Tools [2] *diff* and *cmp*, also performs file comparison based on byte-to-byte analysis. The main difference between *diff* and *cmp* is the output: while *diff* reports whether files are different, *cmp* shows the offsets, line numbers and all characters where compared files differs. *VBinDiff* [3] is another diff-inspired tool that displays the files in hexadecimal and ASCII, highlighting the difference between them. Sandiff also supports byte-level comparison, but the semantic comparison is the main focus of the tool in order to facilitate the testing of large software projects since it allows to discard irrelevant differences in the comparison.

Other approaches to the problem of file comparison, in a *semantic* context, typically use the notion of *change or edit distance* [4] [5]. Wang et. al. [4] proposed X-Diff, an algorithm that analyses the structure of a XML file by applying standard *tree-to-tree* correction techniques that focus on performance. Pawlik et. al. [5] also propose a performance-focused algorithm that is based on the edit distance between ordered labelled nodes of a XML tree. Both approaches can be used by Sandiff to improve its XML-based semantic comparator.

III. Sandiff

Sandiff aims to perform comparison of meaningful data of two artifacts (e.g., directories or files) and report a semantic compatible list that indicates modifications that can impact the output of domain-related on continuous testing setups of large projects. In the context of software testing, syntactically different (byte-to-byte) files can be semantically equivalent. Once the characteristics of a context are defined, previously related patterns to this context can define the compatibility

Configuration 1	*Configuration 2*
ro.build.version.preview_sdk=0	ro.build.version.preview_sdk=0
ro.build.version.codename=REL	ro.build.version.codename=REL
ro.build.version.all_codenames=REL	ro.build.version.all_codenames=REL
ro.build.version.release=8.0.0	ro.build.version.security_patch=17-08-05
ro.build.version.security_patch=17-08-05	ro.build.version.release=8.0.0

Figure 3. Example of AOSP configuration files.

between artifacts from different builds. By definition, two artifacts are compatible when the artifact A can be replace the artifact B without losing its functionality or changing its behavior. As each *file type* has its own set of attributes and characteristics, Sandiff employs specialized semantic comparators that are design to support nontrivial circumstances of domain-specific tests. Consider the comparison of AOSP build output directory and its files. Note that the building process of AOSP in different periods of time can generate *similar* outputs (but not byte-to-byte equivalent). Different byte-to-byte artifacts are called syntactically dissimilar and typically require validation and testing routines. However, on the context where these files are used, the position of *key-value* pairs do not impact testing neither software functionality. We define these files as *semantically compatible*, once Sandiff is able to identify them and suggest a list of tests to skip. Take Figure 3 as example. It shows a difference in the position of the last two lines. When comparing them byte-to-byte, this results in syntactically different files. However, on the execution context where these files are used, this is irrelevant, and the alternate position of lines does not change how the functionality works. Thus, the files are semantically compatible.

Sandiff consists of three main functionalities: (i) input detection, (ii) content recognition, and (iii) specialized semantic comparison, as shown in Figure 2. During analysis of directories and files, we can scan *image files* or *archives* that require particular *read* operations. The first step of Sandiff is to identify these files to abstract *file systems* operations used to access the data. This task is performed by the *Input Recognizer*. Then, the *Content Recognizers* and *Comparators* are instantiated. In order to use the correct *Comparator*, Sandiff implements *recognizers* that are responsible to detect supported *file types* and indicate if a file should be ignored or not based on a test context. Once Sandiff detects a valid file, it proceeds to the semantic comparison. The *Comparators* are specialized methods that take into consideration features and characteristics that are able to change the semantic meaning of execution or testing, ignoring irrelevant syntactically differences. Note that the correct analysis of semantic differences is both *file type* and context sensitive. Sandiff implements two operation

TABLE I. SUMMARY OF *CONTENT RECOGNITION* ANALYSIS FOR EACH FILE.

Attribute	Meaning
Tag	Represents a file type
Action	Action to be taken with the file. (COMPARE *or* IGNORE)
Reason	In case of action IGNORE, the reason of ignore
Context	Information about context that is used to define the ACTION

modes: (i) file and (ii) directory-oriented (walkables). In *file-oriented* mode, the input is two valid comparable files, whereas *directory-oriented* is the recursive execution of *file-oriented* mode in parallel, using a mechanism called *Orchestrator*. In the following sections, we describe the functionalities of Sandiff in detail.

A. *Content Recognition*

To allow the correct selection of *semantic comparators*, Sandiff performs the analysis of file contents by leveraging internal structures and known patterns, i.e., artifact extension, headers, type signatures, and internal rules of AOSP to then summarize the results into (i) tag, (ii) action, (iii) reason, and (iv) context attributes, as shown in Table I. Each attribute helps the *semantic comparators* achieve maximum semantic coverage. To measure the semantic coverage, we gathered the percentage (amount of files) of file types inside vendor.img and created a priority list to develop semantic comparators. For instance, both ELF (32 and 64 bits) files represent about 60% of total files inside .img files, whereas symbolic link files about 14% and XML files about 6%. This process enables us to achieve about 90% of semantic coverage. As the comparison is performed in a semantic mode, it is necessary to know the context in which the artifact was used to enable the correlation between files and test cases. Note that a file can impact one or more tests in a different manner, e.g., *performance*, *security* and *fuzz* tests. The remaining 10% of files are compared using the byte-to-byte comparator.

Each *recognizer* returns a unique tag from a set of known tags, or a tag with no content to indicate that the file could not be recognized. Recognizers can also decide whether a file should be ignored based on context by using the *action attribute* and indicating a justification in the *reason attribute*. Recognizers are evaluated sequentially. The first recognizer runs and tries to tag the file: if the file cannot be tagged, the next recognizer in the list is called, repeating this process until a valid recognizer is found or, in the latter case, the file is tagged to the *default comparator* (byte-to-byte). Table II summarizes the list of AOSP-based recognizers supported by Sandiff.

B. *Semantic Comparators*

Sandiff was designed to maximize semantic coverage of the AOSP by supporting the most relevant intermediate files used for packing artifacts into `.img` image files, i.e., the bootable binaries used to perform factory reset and restore original operational system of AOSP-based devices. To ensure the approach assertiveness, for each semantic comparator, we performed an exploratory data analysis over each file type and use case to define patterns of the context's characteristics. The exploratory data analysis over each file type relies on

TABLE II. LIST OF AOSP-BASED *RECOGNIZERS* SUPPORTED BY SANDIFF.

Recognizer	Tags	Action
IgnoredByContextRecognizer	ignored_by_context	Ingore
ContextFileRecognizer	zip_manifest	Compare
MagicRecognizer	elf, zip, xml, ttf, sepolicy, empty	Compare
AudioEffectsRecognizer	audio_effects_format	Compare
SeappContextsRecognizerc	seapp_contexts	Compare
PKCS7Recognizer	pkcs7	Compare
PropRecognizer	prop	Compare
RegexLineRecognizer	regex_line	Compare
SEContextRecognizer	secontext	Compare
ExtensionRecognizer	Based on file name. e.g.: file.jpg ? ”jpg”	Compare

three steps: (i) file type study, (ii) where these files are used, and (iii) how these files are used (knowledge of its behavior). The result of this analysis was used to implement each semantic comparator. The following subsections describe the main comparators of Sandiff.

1) Checksum: Performs byte-to-byte (checksum) comparison and is the default comparator for binary files (e.g., *bin*, *tlbin*, *dat*) and for cases where file type is not recognized or unsupported. Sandiff employs the industry standard [6] MD5 checksum algorithm, but also offers a set of alternative algorithms that can be set manually by the user: SHA1, SHA224, SHA256, SHA384, SHA512.

2) Audio Effects: AOSP represents audio effects and configurations in `.conf` files that are similar to `.xml`:

(i) `<name>{[sub-elements]}`

(ii) `<name> <value>`

Audio files are analyzed by an ordered model detection algorithm that represents each element (and its sub-elements) as nodes in a tree that is alphabetically sorted.

3) Executable and Linking Format (ELF): ELF files are common containers for binary files in Unix-base systems that packs object code, shared libraries, and core dumps. This comparator uses the definition of the ELF format (`<elf.h>` library) to analyze (i) the files architecture (32 or 64-bit), (ii) the object file type, (iii) the number of section entries in header, (iv) the number of symbols on *.symtab* and *.dynsym* sections, and (v) the mapping of segments to sections by comparing program headers content. To correlate sections to *test cases*, Sandiff detects semantic differences for AOSP *test-sensitive* sections (e.g., *.bss*, *.rodata*, *.symtab*, *.dynsym*, *.text*). When ELF files are Linux loadable kernel modules (.ko extension, kernel object), the comparator checks if the module signature is present to compare its size and values.

4) ListComparator: Compares files structured as list of items, reporting (i) items that exists only in one of the compared files, (ii) line displacements (lines in different positions), and (iv) duplicated lines. To facilitate the correlation between files and *test cases*, Sandiff implements specific semantic comparators for *Prop*, *Regex Line* and *SELinux* files, as they contain properties and settings that are specific to a particular AOSP-based device or vendor.

a) Prop: Supports files with `.prop` extensions and with `<key> = <value>` patterns. Prior to analysis, each line of a *.prop* file is categorized in *import*, *include* or *property*, as defined below:

(i) *import*: lines with format `import <key>`.

(ii) *include*: lines with format `include <key>`.

(iii) *property*: lines with format `<key> = [<value>]`.

After categorization, each line is added to its respective list. The comparator provides a list of properties to be discarded (considered irrelevant) on the semantic comparison. A line can be ignored if is empty or commented.

b) RegexLine: Performs the comparison of files in which all lines match a user-defined regex pattern, e.g., '`/system/.`' or '`.so`', offering the flexibility to perform semantic comparison of unusual files.

c) SELinux: Security-Enhanced Linux, or SELinux, is a mechanism that implements Mandatory Access Control (MAC) in Linux kernel to control the permissions a subject context has over a target object, representing an important security feature for modern Linux-based systems. Sandiff supports semantic comparison of SELinux specification files that are relevant to *security test cases* of the VTS suite, i.e., *Seapp contexts*, *SELinux context*, and *SELinux Policy*, summarizing (i) components, (ii) type enforcement rules, (iii) RBAC rules, (iv) MLS rules, (v) constraints, and (vi) labeling statements.

5) RC: The Android Init System is responsible for the AOSP bootup sequence and is related to the bootloader, *init* and *init* resources, components that are typically customized for specific AOSP-based devices and vendors. The initialization of modern systems consists of several phases that can impact a myriad number of *test cases* (e.g., *kernel*, *performance*, *fuzz*, *security*). Sandiff supports the semantic comparison of `.rc` files that contain instructions used by the *init* system: *actions*, *commands*, *services*, *options*, and *imports*.

6) Symbolic Link: The semantic comparison of symbolic links is an important feature of Sandiff that allows correlation between *test cases* and absolute or relative paths that can be differently stored across specific AOSP-based devices or vendors, but result in the same output or execution. The algorithm is defined as follows: first it checks if the file status is a symbolic link, and if so, reads where it points to. With this content it verifies if two compared symbolic links points to same path. The library used to check the file status depends on the input type and is abstracted by *Input Recognizers*. Take the following instances as examples:

File System → `<sys/stat.h>`

Image File → `<ext2/ext2fs.h>`

ZIP → `<zip.h>`

7) True Type Font: Sandiff uses the Freetype library [7] to extract data from TrueType fonts, which are modeled in terms of faces and tables properties. For each property field, the comparator tags the *semantically* irrelevant sections to ignore during semantic comparison. This is a crucial feature of Sandiff since is common that vendors design different customizations on top of the default AOSP user interface and experience.

8) XML: XML is the *de facto* standard format for web publishing and data transportation, being used across all modules of AOSP. To support the semantic comparison of XML files, Sandiff uses the well-known *Xerces* library [8] by parsing the Document Object Model (DOM), ensuring robustness to complex hierarchies. The algorithm compares nodes and checks if they have (i) different attributes length, (ii) different values, (iii) attributes are only in one of the inputs, and (iv) different child nodes (added or removed).

9) Zip and Zip Manifest: During the building process of AOSP images, zip-based files may contain Java Archives (.jar), Android Packages (.apk) or ZIP files itself (.zip). As these files follows the ZIP structure, they are analyzed by the same semantic comparator. Note that, due to the *archive* nature of ZIP format, Sandiff covers different cases:

(i) *In-place*: there is no need to extract files.

(ii) *Ignore metadata*: ignore metadata that is related to the ZIP specification, e.g., archive creation time and archive modification time.

(iii) *Recursive*: files inside ZIP are individually processed by Sandiff, so they can be handled by the proper *semantic comparator*. The results are summarized to represent the analysis of the zip archive.

Another important class of files of the AOSP building process are the *ZIP manifests*. Manifest files can contain properties that are time-dependent, impacting *naive* byte-to-byte comparators. Sandiff supports the semantic comparison of *manifests* by ignoring *header* keys entries (e.g., String: "Created-By", Regex: "(.+)-Digest") and *files* keys entries (e.g., SEC-INF/buildinfo.xml).

10) PKCS7: Public Key Cryptography Standards, or PKCS, are a group of public-key cryptography standards that is used by AOSP to sign and encrypt messages under a Public Key Infrastructure (PKI) structured as *ASN.1* protocols. To maximize semantic coverage, Sandiff ignores *signatures* and compares only valid *ASN.1* elements.

C. Orchestrator

The *orchestrator* mechanism is responsible to share the resources of Sandiff among a variable number of competing comparison jobs to accelerate the analysis of large software projects. Consider the building process of AOSP. We noticed that, for regular builds, around 384K intermediate files are generated during compilation. In this scenario, running all routines of the official Android Test Suite, known as *Vendor Test Suite* (VTS), can represent a time consuming process that impacts productivity of mobile developers. To mitigate that, the *orchestrator* uses the well-known concept of *workers* and *jobs* that are managed by a priority queue. A *worker* is a thread that executes both recognition and comparison tasks over a pair of files, consuming the top-ranked files in the queue. To accelerate the analysis of large projects, Sandiff adopts the notion of a *fail greedy* sorting metric, i.e., routines with higher probability of failing are prioritized. The definition of *failing priority* is context-sensitive, but usually tend to emphasize critical and time-consuming routines. After the processing of all files, the results are aggregated into a structured report with the following semantic sections: (i) addition, (ii) removal, (iii) syntactically equality, and (iv) semantic equality.

IV. Experiments

In order to verify the comparison performance of Sandiff, we made experiments between different *commercially-available* images of AOSP. The experiments consist on comparing the following image pairs:

TABLE III. OVERALL SUMMARY OF THE IMPACT OF USING SANDIFF IN REAL-WORLD *COMMERCIALLY-AVAILABLE* AOSP BUILDS.

Comparison	**Add**		**Remove**		**Edit**		**Type Edit**		**Equal**		**Error**		**Ignored**	
	Semantinc	Binary	Semantic	Binary	Semantic	Binary	Semantic	Binary	Semantic	Binary	Semantic	Binary	Semantic	Binary
Experiment #1	0	0	0	0	**11**	12	0	0	2185	2185	**0**	19	0	0
Experiment #2	13	13	27	27	0	0	3	3	0	0	0	0	0	0
Experiment #3	23	23	18	18	**527**	606	0	0	**1929**	1805	**0**	45	0	0

* **Add** = file is present on the second input. **Remove** = file is present in the first input. **Edit** = file is present in both inputs, but the comparison returned differences. **Type Edit** = file is present in both inputs, but there were changes in its metadata (e.g., permissions). **Equal** = file is present in both inputs, and the comparison returns an *equal* status. **Error** = file is present in both inputs, but the comparison returns an *error* status. **Ignored** = file is present in both inputs, but is not semantically relevant, so it was ignored.

- **Experiment #1:** Comparing two revisions within same AOSP version: 8.1.0 r64 x 8.1.0 r65.

- **Experiment #2:** Comparing last revision of AOSP Oreo with initial release of AOSP Pie: 8.1.0 r65 x 9.0.0 r1.

- **Experiment #3:** Comparing last revision of AOSP Pie with its initial release: 9.0.0 r1 x 9.0.0 r45.

These pairs were compared using both semantic (Sandiff) and binary (checksum) comparison methods. To evaluate the robustness of each method, we analyzed the files contained in `system.img`, `userdata.img` and `vendor.img` images, which are mounted in the EXT2 file system under a UNIX system. Note that, differently from Sandiff, binary comparison is not capable of reading empty files and symbolic link targets. These files are listed as *errors*, as shown in Table III.

Based on the experiments of Table III, we can note that Sandiff was able to analyze large software projects like the AOSP. First, the semantic comparison was able to determine the file type and to compare not only the file contents, but it is metadata. In contrast, binary comparison was unable to compare symbolic link's targets and broken links failed. Second, the semantic comparison was able to discard irrelevant differences (e.g., the build time in build.prop) which are not differences in terms of functionality. Note that, during *experiment #2*, Sandiff is unable to perform a full analyses between these trees because there were structural changes. For instance, in AOSP Oreo, the `/bin` is a directory containing many files, while in AOSP Pie, the `/bin` is now a symbolic link to another path (that can be another image as well). As a result, Sandiff detects this case as a *Type Edit* and does not traverse `/bin` since it is only a directory in AOSP Oreo.

V. Conclusion

In this paper, we presented Sandiff, a semantic comparator tool that is designed to facilitate continuous testing of large software projects, specifically those related to AOSP. To the best of our knowledge, Sandiff is the first to allow correlation of test routines of the official Android Test Suite (VTS) with semantic modifications in intermediate files of AOSP building process. When used to skip time-consuming *test cases* or to mount a list of priority tests (*fail-fast*), Sandiff can lead to a higher productivity of mobile developers. We showed that semantic comparison is more robust to analyze large projects than binary comparison, since the former is unable to discard irrelevant modifications to the output or execution of the target software. As we refine the semantic comparators of Sandiff, more AOSP specific rules will apply, and consequently, more items can be classified as "Equal" in Sandiff's comparison reports. In the context of making Sandiff domain agnostic, another venue for future work is to explore machine learning techniques to detect how tests are related to different types of files and formats. We also plan to integrate Sandiff to the official Android Test Suite (VTS) to validate our intermediate results.

Acknowledgments

We thank both Rafael Melo da Silva and Nick Diego Yamane Pinto for their valuable help during the development of the project. This work was supported by Sidia: Institute of Science and Technology.

References

[1] Araxis Ltd. Araxis: Software. [Online]. Available: https://www.araxis.com/ [retrieved: October, 2019]

[2] Free Software Foundation, Inc. Diffutils. [Online]. Available: https://www.gnu.org/software/diffutils/ [retrieved: October, 2019]

[3] C. J. Madsen. Vbindiff - visual binary diff. [Online]. Available: https://www.cjmweb.net/ [retrieved: October, 2019]

[4] Y. Wang, D. J. DeWitt, and J. Cai, "X-diff: an effective change detection algorithm for xml documents," in Proceedings 19th International Conference on Data Engineering, March 2003, pp. 519–530.

[5] M. Pawlik and N. Augsten, "Efficient computation of the tree edit distance," ACM Transactions on Database Systems, vol. 40, 2015, pp. 3:1–3:40.

[6] D. Rachmawati, J. T. Tarigan, and A. B. C. Ginting, "A comparative study of message digest 5(MD5) and SHA256 algorithm," Journal of Physics: Conference Series, vol. 978, 2018, pp. 1–6.

[7] FreeType Project. Freetype. [Online]. Available: https://www.freetype.org/freetype2/ [retrieved: October, 2019]

[8] Apache Software Foundation. C xml parser. [Online]. Available: https://xerces.apache.org/xerces-c/ [retrieved: October, 2019]