Build Comparator: Integrated Semantic Comparison for Continuous Testing of Android-based Devices using Sandiff

Carlos E. S. Aguiar, Jose B. V. Filho, Agnaldo O. P. Junior,
Rodrigo J. B. Fernandes, Cícero A. L. Pahins, Paulo C. R. Fonseca
Sidia R&D Institute
Manaus, Brazil
Emails: {carlos.aguiar, jose.vilarouca, agnaldo.junior,
rodrigo.fernandes, cicero.pahins, paulo.fonseca}@sidia.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 solution to efficiently analyze semantic modifications on files that impact domain-specific testing routines of the official Android Test Suite. We also propose the *Build Comparator*, an integrated tool that leverages the *semantic comparison* on *real-world* use cases. We demonstrate our approach by evaluating both *semantic coverage* and scalability on a set of *commercially-available* Android images of a large mobile-related company that comprises both major and minor versions of the system.**

***Keywords*–Testing; Validation; Content Comparison; Continuous Delivery; Tool.**

I. Introduction

As software projects grow up, continuous testing becomes critical, but at the same time, complicated and time-consuming. Consider a project with a million files and intermediate artifacts. A *test suite* that offers *continuous testing* functionalities must perform without creating bottlenecks or impacting project deliveries. However, effectively using continuous integration can be a problem: tests are time-consuming to execute. Consequently, 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 evaluation 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 testing and evaluation process 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 computing and time-intensive tasks.

Sandiff [1] 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 generated in different moments, but with the same source code, create the environment and build instructions, the final result is different in byte level (byte-to-byte). Still, it can be semantically equivalent based on its context (meaning). In this case, it is expected that these two builds perform the same. However, how to guarantee this? Our solution relies on how to compare and demonstrate that two AOSP builds are semantically equivalent. 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 Vendor Test Suite (VTS). As our solution provides a list of semantically equivalent files, it is possible to skip tests that show the behavior provided by these files. The Figure 1 shows the execution official Android Test Suite is execute in a *commercially-available* build based on AOSP. The execution of all modules exceeded 4 hours, compromising developer performance and deliveries on a planned schedule.

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 large software. By *large*, we mean software designed by a vast number of developers inserted in a distributed software development environment; after that, automatic test suits are necessary.

Another motivation of Sandiff is to enable developers to find bugs faster on complex software. Take as an example a camera bug in which the component or module is part of a complex Android subsystem stack covering various architectural levels: application framework, vendor framework, native code, firmware, and others. With the help of the Sandiff, a developer can analyze the semantic comparison between different software releases and narrow the source of the problem.

In summary, we present the key research contributions of our proposal:

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.

1. An approach to perform *semantic* comparison and facilitate *continuous testing* of large software projects.
2. An integrated *Build Comparator* tool that leverages semantic comparison to support Android-based software releases and DevOps teams.
3. 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 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 *Build Comparator* tool and its architecture, which enables the integration of Sandiff on systems of the Android development environment. In Section V, 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 VI.

II. Related Work

To the best of our knowledge, few literature approaches propose *comparison* of files with different formats and types. Most comparison tools focus on the comparison based on diff (text or, at most, byte position). Araxis [2] 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 [3] *diff* and *cmp*, also perform 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* [4] is another diff-inspired tool that displays the files in hexadecimal and ASCII, highlighting the difference between them.

Other approaches to the problem of file comparison, in a *semantic* context, typically use the notion of *change or edit distance* [5] [6]. Wang et al. [5] proposed X-Diff, an algorithm that analyses the structure of an XML file by applying standard *tree-to-tree* correction techniques that focus on performance. Pawlik et al. [6] also propose a performance-focused algorithm based on the edit distance between ordered labeled nodes of an XML tree. Both approaches can be used by Sandiff to improve its XML-based semantic comparator. Similarly, with study applied on music notations, in [7] implemented a solution to compare files, like XML, based on a tree representation of music notation combining with sequence and tree distance. Besides, the authors show a tool to visualize the differences side-by-side.

In [8], is showed a tool named Diffi, which is a diff improved build to observe the correlation between file formats. So, Diffi verifies the heaps of the files' reflection levels and discovers which file levels can recognize the delta between two files correctly.

In [9], the authors explore the use of wavelets for the division of documents into fragments of various entropy levels, which made a separation between grouping sections to decide the similarity of the files, and finally, detect malicious software. Additionally, with a similar objective in [10] is investigated the applicability of machine learning techniques for recognizing criminal evidence in activities into file systems, verifying possible manipulations caused by different applications.

Different from previous works, the Sandiff also supports byte-level and semantic comparison simultaneously. However, the semantic comparison is the main focus of the tool to facilitate extensive software projects testing since it allows to discard irrelevant differences in the comparison.

III. Sandiff

Sandiff aims to compare meaningful data of two artifacts (e.g., directories or files) and report a compatible semantic 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 between artifacts from different builds. By definition, two artifacts are compatible when the artifact A can replace the artifact B without losing its functionality or changing their behavior. As each *file type* has its own set of attributes and characteristics, Sandiff employs specialized semantic comparators that are designed 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, in the context where these files are used, the position of *key-value* pairs do not impact testing either 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, in 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

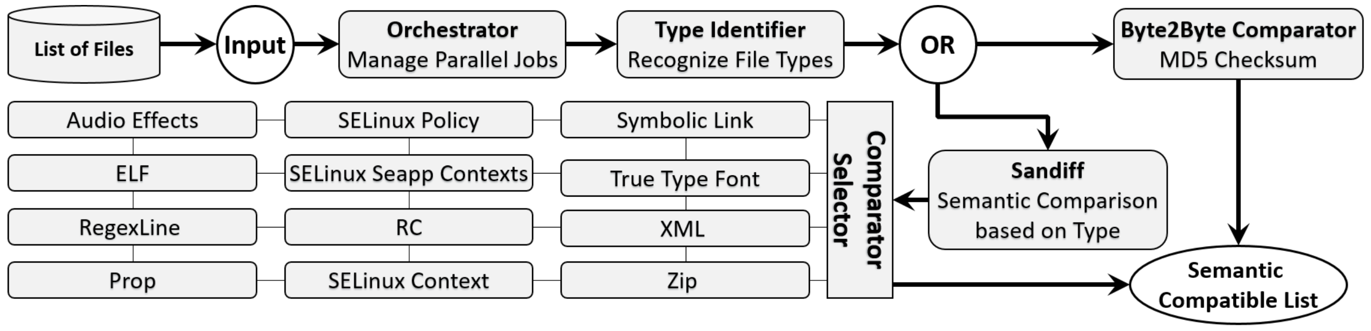


Figure 2. Sandiff verifies the semantic compatibility of two files or directories (list of files) and report their differences.

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.

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

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 syntactical differences. Note that the correct analysis of semantic differences is both *file type* and context-sensitive. Sandiff implements two operation 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

The Sandiff performs the analysis of file contents by leveraging internal structures, and known patterns to allow the correct selection of *semantic comparators*, 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* to achieve maximum semantic coverage over file types inside images embedded on commercially-available devices. The Android-based devices include several partitions that serve for specific purposes on boot process and general operation, as defined below:

- `system.img`: contains the Android operating system framework.
- `vendor.img`: contains any manufacturer-specific binary file that is not available to the AOSP.
- `userdata.img`: contains user-installed data and applications, including customization data.

To measure the semantic coverage, we gathered the percentage (amount of files) of file types inside `vendor.img`. We created a priority list to develop semantic comparators, as shown in the Table II. For instance, both ELF (32 and 64 bits) files represent about roughly 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 form, 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 available 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 III 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 a factory reset and restore the original operating system of AOSP-based devices. To ensure

TABLE II. SUMMARY OF SANDIFF SEMANTIC COVERAGE.

File Type	# Files	Percentage (%)
🟩 ELF-64	320	31.34
🟩 ELF-32	298	29.19
🟩 Symbolic link	152	14.89
🟩 XML document text	63	6.17
🟩 RC	34	3.33
🟥 .bin	34	3.33
🟥 .tlbin	28	2.74
🟩 .prop	21	2.06
🟩 .conf	10	0.98
🟩 ASCII text	8	0.79
🟩 Exported SGML	6	0.59
🟥 .dat	6	0.59
🟧 .hcd	4	0.39
🟩 JAR	3	0.29
🟧 .txt	3	0.29
🟩 .xml	2	0.20
🟩 Gzip compressed data	1	0.10
🟩 SE Linux	1	0.10
🟩 PEM certificate	1	0.10

* **Green** = Semantic comparison is performed. **Red** = Semantic comparison not applicable. Default comparator (checksum) is performed. **Orange** = Semantic comparison not supported by Sandiff. Default comparator (checksum) is performed.

TABLE III. 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	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

the approach assertiveness, we performed an exploratory data analysis over each file type and use case to define patterns of the context's characteristics for each semantic comparator. The exploratory data analysis over each file type relies on three steps:

1. file type study;
2. where these files are used;
3. 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 semantic comparators of Sandiff.

1) Default (Fallback) Comparator: Performs byte-to-byte (checksum) comparison and is the default comparator for binary files (e.g., *.bin*, *.tlbin*, and *.dat*). Acts as a *fallback alternative*, performing comparison for cases where (i) file type is not recognized or is unsupported and (ii) due to any errors during the comparison (e.g., corrupted or malformed files). Sandiff employs the industry standard [11] MD5 checksum algorithm, as it balances performance and simplicity to verify data integrity. For security-critical scenarios, Sandiff offers a set of *SHA* algorithms with improved robustness and reliability for collision attacks, i.e., verification of intentional corruption, despite lower run-time performance. The supported alternatives are: SHA1 [12], SHA224 [13], SHA256 [14], SHA384 [15], and SHA512 [16]. To choose the most suitable algorithm, it is necessary to consider the infrastructure, the application requirements and the knowledge of the developers. A complete overview and discussion about SHA algorithms is provided in [17], [18] reviews.

TABLE IV. EXAMPLE OF COMPARING TWO SEQUENCES OF BYTES.

Position	0	1	2	3	4	5	6	7	8
Sequence #1	1D	E1	2A	DD	5F	AE	F8	5F	19
Sequence #2	1D	ED	31	9E	5F	08	F8	5F	2E

When comparing two or more sequences of bytes, each position is represented on hexadecimal format, i.e., positional format that represents numbers using a base of 16 and uses symbols `0-9` and `A-F`, representing each byte by two hexadecimal digits. Table IV illustrates how Sandiff performs the byte-to-byte comparison. Note that in this case, the comparison verified that positions 1-3, 5 and 8 are different, summarizing the result as illustrated in Listing 1.

```
Difference(s):
    Differs on range between position 1 and 3
    Differs on byte at position 5
    Differs on byte at position 8
```

Listing 1. Example of checksum comparator output.

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 analysed by an ordered model detection algorithm that represents each element (and its sub-elements) as nodes in a tree 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 analyse (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*), listing and performing byte-to-byte comparison on all relevant ELF sections found on target files. Table VI summarizes irrelevant sections to ignore when implementing the semantic comparison for ELF files. When ELF files are Linux loadable kernel modules (`.ko` extension,

TABLE V. LIST OF LISTCOMPARATOR OPERATIONS MODES AND FLAGS SUPPORTED BY SANDIFF.

Flag	Description
File Order	Indicates if there are difference among the item position in list and the line number of file
Line Type	Indicates if file lines has only one key ONLY_KEY) or are (composed by more elements (MORE_THAN_KEY), with a value associated to the key
Displacement	Indicates if the result refers to differences semantically relevant (true), or differences semantically irrelevant (false)

TABLE VI. LIST OF IRRELEVANT SECTIONS FOR SEMANTIC COMPARISON OF ELF FILES.

Section	Reason
.debug_*	holds information for symbolic debugging
.comment	version control information
.gnu_debugdata	allows adding a subset of full debugging info to a special section of the resulting file so the stack traces can be more easily "symbolicated"
.gnu_debuglink	contains a file name and a CRC checksum of a separated debug file
.gnu_hash	allow fast lookup up of symbols to speed up linking
.ident	where GCC puts its stamp on the object file, identifying the GCC version which compiled the code
.shstrtab	section names
.got.*	provides direct access to the absolute address of a symbol without compromising position-independence and sharebility
.note.gnu.build-id	unique identification 160-bit SHA-1 string

kernel object), the comparator checks if the module signature is present to compare its size and values. Signature differences are not considered relevant to semantic comparison. In case of any ELF file is corrupted or malformed, the *fallback* comparison is performed.

4) ListComparator: Base comparator for files structured as item lists, reporting (i) items that exist only in one of the compared files, (ii) line displacements (i.e., lines in different positions), (iii) comments and empty lines, and (iv) duplicated lines. To facilitate the correlation between files and *test cases*, Sandiff implements specific *list-based* 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. To support such variety of files, the *list-based* comparators offers a list of operation modes to tackle specific scenarios, e.g., when the file has empty lines or comments semantically irrelevant, as summarized in Table V. The following paragraphs describe the *list-based* comparators of Sandiff.

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

1. *import*: lines with format `import <key>`, i.e., lines containing the word "import", followed by one key.
2. *include*: lines with format `include <key>`, i.e., lines containing the word "include", followed by one key.
3. *property*: lines with format `<key> = [<value>]`, i.e., lines containing a pair composed by a key and an associated value (optional) - separated by "=" symbol.

After categorization, each line is parsed and added to its respective list, i.e., *import*, *include*, and *property* lists. Each of the three lists is individually compared with the others, generating disjoint results that are later jointed for reporting.

TABLE VII. LIST OF SEMANTIC IRRELEVANT PROPERTIES OF ANDROID BUILDS.

Property	Description
BD	Found in system/sepolicy_version
ro.bootimage.build.*	Build property set by android/build/make/core/Makefile
ro.build.*	Build property set by android/build/make/tools/buildinfo.sh at each new build
ro.expect.recovery_id	Build property set by android/build/make/core/Makefile
ro.factory.build_id	Build property set by android/build/make/core/main.mk
ro.ss.bid	Build property set by android/build/make/core/main.mk
ro.vendor.build.*	Build property set by android/build/make/core/Makefile

The *PropComparator* also provides a list of properties to be discarded (considered irrelevant) on the semantic comparison, as summarized in Table VII. 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, *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:

- *imports*: analyses the importing calls of the invoking mechanism.
- *actions*: compares the sequence of `commands` to test if logical conditions are met, since commands in different order may lead to different results.
- *services*: tackles operation modes by analysing options (e.g., *oneshot*, *onrestart*, etc.) and characteristics (e.g., critical, priority, etc.) of the *init-related* services.

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*. The libraries are:

$$\text{File System} \rightarrow \texttt{<sys/stat.h>}$$
$$\text{Image File} \rightarrow \texttt{<ext2/ext2fs.h>}$$
$$\text{ZIP} \rightarrow \texttt{<zip.h>}$$

7) True Type Font: Sandiff uses the Freetype library [19] 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 customization 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 [20] 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 follow the ZIP structure, they are analysed by the same semantic comparator. Note that, due to the *archive* nature of ZIP format, Sandiff covers different cases:

1. *In-place*: there is no need to extract files.
2. *Ignore Metadata*: ignore metadata that is related to the ZIP specification, e.g., archive creation time and archive modification time.
3. *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`).

Each *APK* – the package file format used by Android – has a set of manifest information and other metadata that are part of the signature process. The most relevant for semantic comparison are `META-INF/CERT.SF` and `MANIFEST.MF` files, since it contains integrity checks for files which are included in the distribution, as shown in Listings 2 and 3. As both files share the same structure, it is possible to analyse them with the same semantic comparator. The *ZipManifestComparator* parses both files and compares headers and digests by ignoring irrelevant header and files entries.

```
Manifest-Version: 1.0
Created-By: singlejar
Name: AndroidManifest.xml
SHA-256-Digest: cEnjm4r95tb8NSMCP6B2Nn+P1G8sIpeXpPtsmuvSnfM=
Name: META-INF/services/
     com.google.protobuf.GeneratedExtensionRegistryLoader
SHA-256-Digest: AT7RUk9qflHB8mVVceY0Zi7UuRK2bIPMewdxqL2zIBY=
Name: android-support-multidex.version.txt
SHA-256-Digest: OuJR1NnX1srJFP8Td2Bv9F5nMX3O5iAgxf15egCfa+Q=
```

Listing 2. The `MANIFEST.MF` file contains metadata used by the java run-time when executing the *APK*.

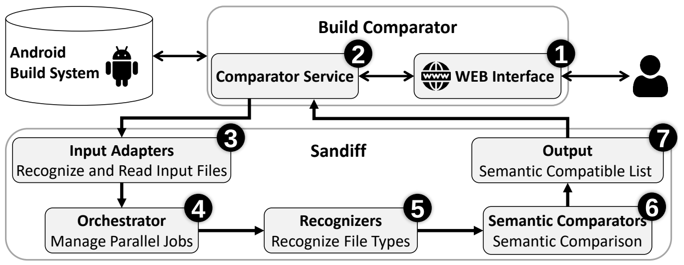


Figure 4. *Build Comparator* architecture and its relation with Sandiff's semantic comparison features.

```
Signature-Version: 1.0
Created-By: 1.0 (Android SignApk)
SHA-256-Digest-Manifest: rk0ZXezaawnGF65RmyYEmpqL+
     gFdHzRTNb3kr/NeNNQ=
X-Android-APK-Signed: 2, 3
Name: AndroidManifest.xml
SHA-256-Digest: ZgWkXiulhWzT7qwbAVYgepd5tyGt6D+RQNAeT+AJw1Y=
Name: META-INF/services/
     com.google.protobuf.GeneratedExtensionRegistryLoader
SHA-256-Digest: ASo5NB1Aa4gclvZke+olzjfErZMzxn/hthDK7Ann56w=
Name: android-support-multidex.version.txt
SHA-256-Digest: 6/lnFOH7mFVER94rAWcUmubglFFrHR7nf8+7zqQOgQs=
```

Listing 3. The `CERT.SF` file contains the whole-file digest of the `MANIFEST.MF` and its sections.

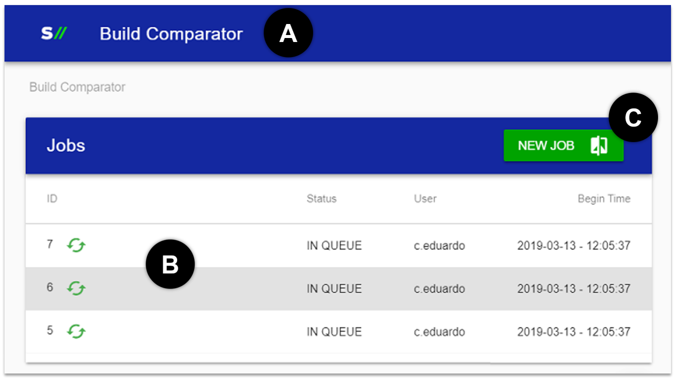
10) PKCS7: Public Key Cryptography Standards, or PKCS, are a group of public-key cryptography standards that is used by AOSP to sign and encrypt messages under a Public Key Infrastructure (PKI) structured as *ASN.1* protocols. To maximize semantic coverage, Sandiff ignores *signatures* and compares only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

To provide *semantic comparison* benefits on supporting software releases and DevOps operations, we propose *Build Comparator*, an integrated tool that abstracts configuration, image management, test execution, and report visualization to



Build Comparator

Build Comparator

Jobs

ID	Status	User	Begin Time
7	IN QUEUE	c.eduardo	2019-03-13 - 12:05:37
6	IN QUEUE	c.eduardo	2019-03-13 - 12:05:37
5	IN QUEUE	c.eduardo	2019-03-13 - 12:05:37

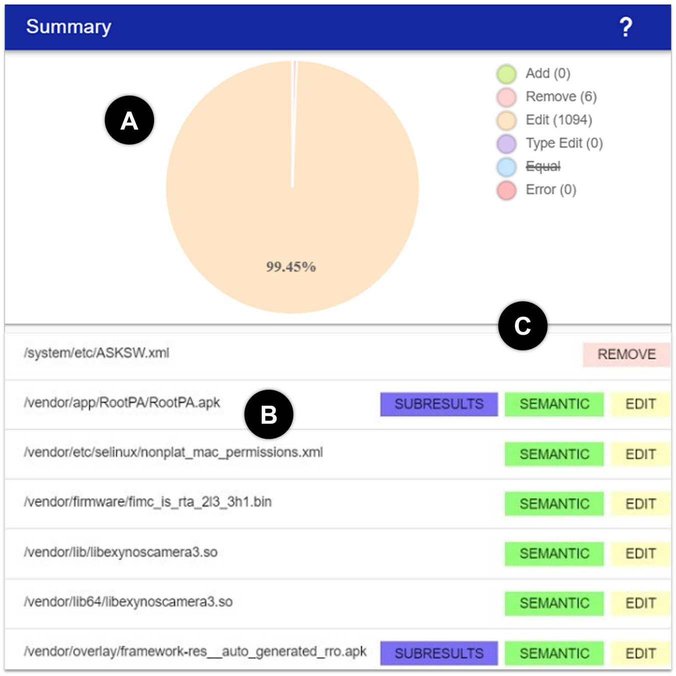
Figure 5. **(A)** Main *Build Comparator* interface. **(B)** List of jobs with unique identifier (ID), status, user, and job begin time. **(C)** Creation of comparison jobs based on (i) build systems, (ii) remote or (iii) local images.

facilitate Android-based development pipelines and processes. The *Build Comparator* can be integrated with systems that are commonly used for compilation jobs and managing the configuration and execution of Android's platform scripts, as shown in Figure 4. The proposed integration covers all main steps from user input to report visualization, as follows:

1. The user interacts with the tool through the **Web Interface**, which is responsible for providing communication between the user and service.
2. The **Comparator Service** provides a REST-based service that is responsible for managing the whole life-cycle of scheduled jobs (i.e., creation, management and running) using Sandiff as back-end. The *service* is also integrated with the systems responsible for compilation jobs, performing the user credential and image downloads.
3. On Sandiff side, the **Input Adapters** provide the ability to R/W different input files, abstracting the methods used to access the data.
4. The **Orchestrator** is responsible for managing the parallel jobs when Sandiff is filled with directories analysis, tracking file additions, removals, and type changes.
5. The **Recognizers** are responsible for determining the correct **Semantic Comparator** by analysing the (i) file type, (ii) header, and (iii) general structure.
6. In the last step, Sandiff generates the *semantic* compatible list, making it available to the *Comparator Service* and *Web Interface*.

In Figures 5 and 6, we summarize the main *Build Comparator* interfaces. We use a client-server architecture for the current implementation. The browser-based client is written in *JavaScript*, *HTML5*, and *Angular*. The server is a C++11 template-based implementation that exposes its API for queries via *HTTP*, enabling the comparison of Android builds according with the source of the artifacts. The architecture supports the most common integration methods with *continuous testing* tools or development pipelines:

- **Build Systems:** the artifacts are located in continuous integration and deployment servers, e.g., *QuickBuild* [21].
- **Remote:** the artifacts are located in an *HTTP* server or the cloud.
- **Local:** the artifacts are located in the user's personal computer.



Summary

🟢	Add (0)
🔴	Remove (6)
🟠	Edit (1094)
🟣	Type Edit (0)
🔵	~~Equal~~
🔴	Error (0)

/system/etc/ASKSW.xml			REMOVE
/vendor/app/RootPA/RootPA.apk	SUBRESULTS	SEMANTIC	EDIT
/vendor/etc/selinux/nonplat_mac_permissions.xml		SEMANTIC	EDIT
/vendor/firmware/fimc_is_rta_2l3_3h1.bin		SEMANTIC	EDIT
/vendor/lib/libexynoscamera3.so		SEMANTIC	EDIT
/vendor/lib64/libexynoscamera3.so		SEMANTIC	EDIT
/vendor/overlay/framework-res__auto_generated_rro.apk	SUBRESULTS	SEMANTIC	EDIT

Figure 6. Interface that summarizes the *semantic comparison* results. **(A)** General statistics of the analysed pair of Android images. **(B)** List of semantic relevant artifacts that are supported by Sandiff. **(C)** Type of each semantic modification (add, remove, edit, type edit).

V. Experiments

A. Semantic Coverage

To verify the comparison performance of Sandiff, we did experiments between different branches of *commercially-available* images of AOSP. The AOSP contains numerous types of files (text, audio, video, symbolic links, binary files, among others) that can be compared semantically. The experiments consist of comparing the following image pairs:

- **Experiment #1:** Analysing two major AOSP with minor revisions: 8.1.0 r64 x 8.1.0 r65.

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

- **Experiment #3:** Analysing the last revision of AOSP Pie and its initial release: 9.0.0 r1 x 9.0.0 r45.

- **Experiment #4:** Analysing two major releases of AOSP Oreo and AOSP 10: 8.1.0 r77 x 10.0.0 r39.

- **Experiment #5:** Analysing two major releases of AOSP Pie and AOSP 10: 9.0.0 r57 x 10.0.0 r39.

These pairs were compared using both semantic (Sandiff) and byte-to-byte (checksum) comparison methods. To demonstrate the robustness of each method, we analysed 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, the byte-to-byte comparison cannot read empty files and symbolic link targets. These files are listed as *errors*, as shown in Table VIII.

Based on the experiments of Table VIII, we can note that Sandiff was able to analyze large software projects like the AOSP. First, the semantic comparison was able to determine

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

Comparison	**Add**		**Remove**		**Edit**		**Type Edit**		**Equal**		**Error**		**Ignored**	
	Semantic	Binary	Semantic	Binary	Semantic	Binary	Semantic	Binary	Semantic	Binary	Semantic	Binary	Semantic	Binary
Experiment #1	0	0	0	0	11	12	0	0	2185	2165	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
Experiment #4	179	179	41	41	98	97	5	5	153	154	0	0	0	0
Experiment #5	439	439	240	240	839	844	11	11	785	780	0	0	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.

the file type and compare the file contents and its metadata. In contrast, a byte-to-byte comparison was unable to compare the symbolic link's targets and broken links. Second, the semantic comparison was able to discard irrelevant differences (e.g., the build time in `build.prop`) which are no differences in functionality.

Note that, during *experiment #2*, Sandiff is unable to perform a full analysis between these trees because there were structural changes. For instance, in AOSP Oreo, the `/bin` is a directory containing many files. In contrast, in AOSP Pie, the `/bin` is now a symbolic link to another directory path (that can be another image). 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.

The *experiment #3* is similar to *experiment #1*, except that the number of edited files is significantly more extensive since the code has changed due to the different revisions. We notice that *errors* occur in symbolic links, as expected for byte-to-byte comparison. Some files only changed in terms of data, but not in semantic meaning, making this the optimal scenario for Sandiff over the traditional checksum.

Both *experiments #4* and *#5* evaluate at which point the semantic comparison becomes irrelevant, i.e., they exploit Sandiff performance when analyzing significantly different AOSP releases. As expected for these scenarios, the results of both semantic and byte-to-byte comparisons are similar. In summary, the *semantic comparison* is inaccurate when analyzing files that are not recognized by the rules in the current implementation of Sandiff, making the byte-to-byte more appropriate in these cases. Nevertheless, Sandiff is able to support both *semantic* and *non-semantic* tasks, despite run-time performance disadvantages when compared with naive solutions.

B. Scalability

To study the behavior of Sandiff when dealing with multi-threaded AOSP build systems, we performed a scalability evaluation that measures the run-time performance on different (i) execution modes, (ii) number of concurrent *comparison* jobs, and (iii) AOSP builds. In this experiment, we used a workstation-based setup with an Intel Core i7-2600 at 3.40GHz with 16GB of memory, hereafter called *Machine #1*, and a data center server with an Intel Xeon E5-2697 at 2.30GHz with 125GB of memory, hereafter called *Machine #2*.

The *execution modes* are responsible for defining the parallel and recursive operations of Sandiff's *Orchestrator*, as defined in Section III. In summary, it manages the strategies for resource sharing and how comparison results are collected. Below we list the evaluated modes:

- **Walk First:** leverages multi-threading by sequentially analyzing the directories, distributing its files across the comparison jobs. It is the default mode of the Sandiff.
- **Parallel Walk:** performs concurrent directory analysis up to the number of comparison jobs. It is the recommended mode for analyzing directories with a large number of files.
- **Mixed:** iterates in both files and directories.
- **Slice:** lists all files before processing, then distributes them in similar batches across the maximum number of comparison jobs.

To minimize the variance between runs, we repeated each experiment four times, as shown in Figures 7 and 8. Note that, despite different scenarios, Sandiff achieved its best run-time performance when running with four parallel comparison jobs. Due to AOSP nature, Sandiff cannot successfully parallelize the jobs since the tasks are interdependent. In general, *Walk First*, *Parallel Walk*, and *Mixed* modes tend to attain similar scalability.

To cope with the variance, we repeated each experiment four times, as shown in Figures 7 and 8. Note that, despite different scenarios, run-time performance increases quickly as the number of jobs grows. Due to the small amount of data in pure AOSP images - not more than 1900 files and 800MB - the orchestration process among multiple jobs creates an overhead that is not compensated by the parallelization after four parallel jobs. This limitation is overcome when larger images are used. To illustrate the scenario where more data is compared, we run a comparison between two commercially-available builds based on AOSP having 4847 files and a total size of 4GB. As can be seen in Figure 9, parallel comparison stands up when a higher amount of data is compared. Execution time decreases as the number of jobs gets closer to the number of physical cores available on the machine.

In general, *Walk First*, *Parallel Walk*, and *Mixed* modes tend to attain similar scalability, but *Slice* mode provided better performance, relatively and absolutely, when the number of jobs coincides with the number of available cores of the machine. This occurs due to the decreased number of context changes provided by the Slice mode combined with the maximum usage of available cores.

VI. Conclusion

In this paper, we presented *Build Comparator*, an integrated tool for supporting software releases and DevOps operations

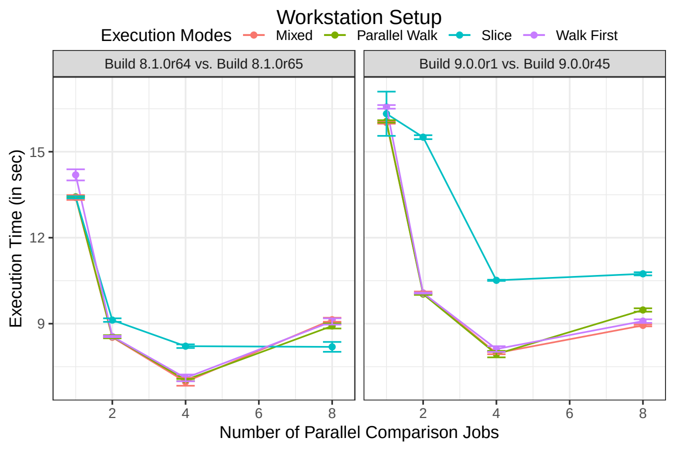


Figure 7. Scalability performance when analysing AOSP builds on workstation-based setups.

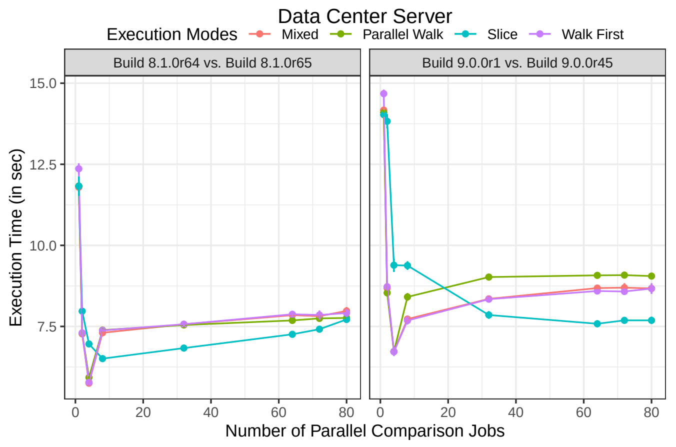


Figure 8. Scalability performance when analysing AOSP builds on data center servers, i.e., dedicated environments for experimenting.

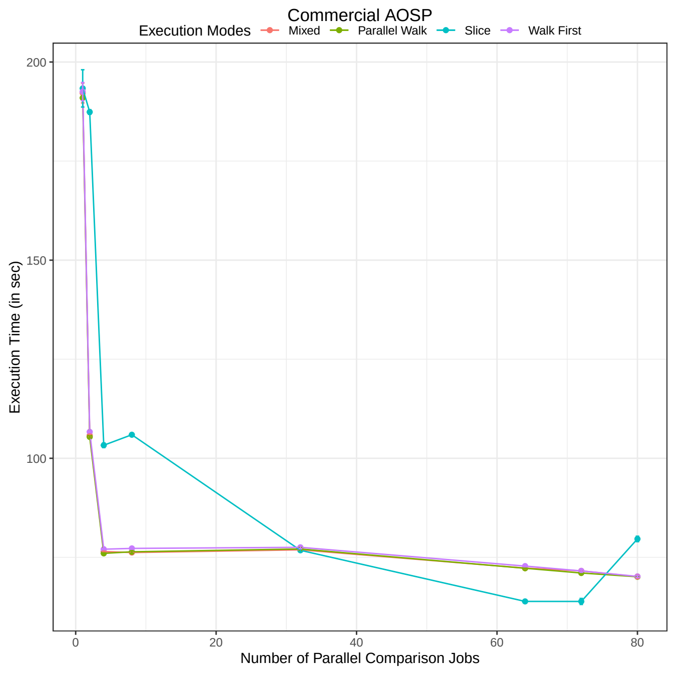


Figure 9. Scalability performance when analysing a large commercial AOSP build on a data center server.

on Android development pipelines by leveraging Sandiff, a semantic comparator 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 higher productivity of mobile developers. We showed that semantic comparison is more robust to analyze large projects than binary comparison since the latter cannot discard irrelevant modifications to the target software's output or execution. 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.

With *Build Comparator*, we presented and analyzed an architecture that enables the integration of *semantic comparison* with systems that are commonly used in the development of AOSP software, exploiting *real-world* use cases. In the context of making Sandiff domain agnostic, another avenue for future work is to explore machine learning techniques to detect how tests are related to different files and formats. We also plan to extend *Build Comparator*'s reporting features by proposing visualizations that highlight relevant semantic differences between pairs of files and 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 partially supported by Samsung Eletrônica da Amazônia Ltda., under the auspice of the informatics law no 8.387/91.

REFERENCES

[1] C. E. D. S. Aguiar, J. I. B. V. Filho, A. O. P. Junior, R. J. B. Fernandes, and C. A. D. L. Pahins, "Sandiff: Semantic file comparator for continuous testing of android builds," VALID 2019 : The Eleventh International Conference on Advances in System Testing and Validation Lifecycle, Nov. 2019, pp. 51–55.

[2] Araxis Ltd. Araxis: Software. [Online]. Available: https://www.araxis.com/ [retrieved: November, 2020]

[3] Free Software Foundation, Inc. Diffutils. [Online]. Available: https://www.gnu.org/software/diffutils/ [retrieved: October, 2020]

[4] C. J. Madsen. Vbindiff - visual binary diff. [Online]. Available: https://www.cjmweb.net/ [retrieved: October, 2020]

[5] 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.

[6] 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.

[7] F. Foscarin, F. Jacquemard, and R. Fournier-S'niehotta, "A diff procedure for music score files," in 6th International Conference on Digital Libraries for Musicology, ser. DLfM '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 58–64. [Online]. Available: https://doi.org/10.1145/3358664.3358671

[8] G. Barabucci, “Diffi: Diff improved; a preview,” in Proceedings of the ACM Symposium on Document Engineering 2018, ser. DocEng ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3209280.3229084

[9] I. Sorokin, “Comparing files using structural entropy,” Journal in Computer Virology, vol. 7, 2011. [Online]. Available: https://doi.org/10.1007/s11416-011-0153-9

[10] R. M. A. Mohammad and M. Alqahtani, “A comparison of machine learning techniques for file system forensics analysis,” Journal of Information Security and Applications, vol. 46, 2019, pp. 53 – 61. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2214212618307579

[11] 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.

[12] G. Leurent and T. Peyrin, “From collisions to chosen-prefix collisions application to full sha-1,” in Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, 2019, pp. 527–555.

[13] R. Martino and A. Cilardo, “Sha-2 acceleration meeting the needs of emerging applications: A comparative survey,” IEEE Access, vol. 8, 2020, pp. 28 415–28 436.

[14] D. M. A. Cortez, A. M. Sison, and R. P. Medina, “Cryptographic randomness test of the modified hashing function of sha256 to address length extension attack,” in Proceedings of the 2020 8th International Conference on Communications and Broadband Networking, 2020, pp. 24–28.

[15] P. M. Simanullang, S. Sinurat, and I. Saputra, “Analisa metode sha384 untuk mendeteksi orisinalitas citra digital,” KOMIK (Konferensi Nasional Teknologi Informasi dan Komputer), vol. 3, no. 1, 2019.

[16] A. Jose and K. Subramaniam, “Dna based sha512-ecc cryptography and cm-csa based steganography for data security,” Materials Today: Proceedings, 2020.

[17] S. Long, “A comparative analysis of the application of hashing encryption algorithms for MD5, SHA-1, and SHA-512,” Journal of Physics: Conference Series, vol. 1314, Oct. 2019, p. 012210.

[18] G. Shaheen, “A robust review of sha: Featuring coherent characteristics,” International Journal of Computer Science and Mobile Computing, vol. 9, 2020, p. 111–116.

[19] FreeType Project. Freetype. [Online]. Available: https://www.freetype.org/freetype2/ [retrieved: November, 2020]

[20] Apache Software Foundation. C xml parser. [Online]. Available: https://xerces.apache.org/xerces-c/ [retrieved: October, 2020]

[21] PMEase. Quickbuild. [Online]. Available: https://www.pmease.com/ [retrieved: October, 2020]