Automatic Signature Generation for Anomaly Detection in Business Process Instance Data

Kristof Böhmer and Stefanie Rinderle-Ma

University of Vienna, Faculty of Computer Science
{kristof.boehmer,stefanie.rinderle-ma}@univie.ac.at

Abstract. Implementing and automating business processes often means to connect and integrate a diverse set of potentially flawed services and applications. This makes them an attractive target for attackers. Here anomaly detection is one of the last defense lines against unknown vulnerabilities. Whereas anomaly detection for process behavior has been researched, anomalies in process instance data have been neglected so far, even though the data is exchanged with external services and hence might be a major sources for attacks. Deriving the required anomaly detection signatures can be a complex, work intensive, and error-prone task, specifically at the presence of a multitude of process versions and instances. Hence, this paper proposes a novel automatic signature generation approach for textual business process instance data while respecting its contextual attributes. Its efficiency is shown by an comprehensive evaluation that applies the approach on thousands of realistic data entries and $240,000$ anomalous data entries.

Keywords: Anomaly Detection, Process Instance, Regex, Textual Data

1 Introduction

Business processes have risen to important and deeply integrated solutions which spawn over various organizations and interconnect a multitude of different services and applications [4]. Hence, ensuring business process security is a crucial challenge [8]. To address this challenge, process models can be interpreted as networks. They connect, for example, legacy applications [13] that were originally not intended to be globally linked or services that are not controlled by the process owner and, therefore, should not be trusted. Although the vulnerability of IT supported business process models is generally accepted [8], we found that the business process security monitoring area is still underdeveloped compared to “classic” IT network security [9].

This surprises because the importance and widespread automated execution of processes makes them an attractive target for attackers [10]. Two different scenarios can occur: In the *first*, a *targeted attack* is executed whereby the attacker has in-depth knowledge of the attacked process model. Such attacks are difficult to prevent. However, to prepare a targeted attack, the attacker must *probe* the process to identify vulnerabilities—which is the *second* scenario.

Table 1. Generate signatures automatically from recorded data.

No.	Data Type	Analyzed Process Instance Data	Matches Signature
1	Normal	‘a=mapred:8␣set␣PCMA/8000’	— (Signature not yet generated)
2	Normal	‘a=mapred:3␣startFAPCM␣GSM/8000’	— (Signature not yet generated)
3	Normal	‘a=mapred:0␣test␣PCMU/8000’	— (Signature not yet generated)
4	Normal	‘a=mapred:3␣hello␣GSM/8000’	— (Signature not yet generated)
5	Normal	‘a=mapred:5␣stop␣GSM/8000’	— (Signature not yet generated)
6	*Generated Signature*	‘^(a=mapred:)(\d\s\w\w\w)(.){0,7} (␣PCMA\/8000|␣GSM\/8000|␣\/8000)$’	*Generated by the presented approach.*
7	Normal	‘a=mapred:4␣reject␣PCMA/8000’	Yes
8	Attacker	‘a=attack:0␣8D␣14␣03␣PCMU/8000’	*No* → Anomaly detected

We assume that probing and attacks deviate from *normal use*. For example, under normal use data values that trigger potential buffer overflow vulnerabilities, remain, likely, unobserved. Hence, such data values can be detected as anomalies, which are, events with relatively small probabilities of occurrence [1]. If an anomaly, indicating probing or attacks, can be detected in advance, then the affected process instances could, for example, be halted or migrated to a honey pot and the attack, thereby, be prevented.

A common approach to detect anomalies is to define signatures which represent the expected typical behavior [16]. Hence, if the signatures do not match the observed behavior then an anomaly was detected. Compare with Table 1—which is also employed as a *running example*. Multiple normal (i.e., expected structure/ type/content) variable values[1] (*No. 1-5*, i.e., Analyzed Process Instance Data) are used to create a signature *(No. 6)*. This signature identifies a probing message from the attacker *(No. 8)* as an anomaly and, therefore, as a potential attack.

However, creating such signatures manually is time-consuming and error-prone because an enormous amount of frequently changing complex business processes [4] is currently in use in large scale systems [12]. Hence, an *automatic signature generation approach* is proposed in the following.

Additionally, we found that existing process anomaly detection work, cf. [2, 1, 11, 3, 16], is not capable of analyzing arbitrary *textual process instance data* values (i.e., string variables holding, e.g, XML, JSON, or EDIFACT data formats along with dates, booleans, or exchanged messages). This limitation is critical when considering that today's business processes frequently utilize textual variables to flexibly store and process various kinds of information. Moreover, existing work, considers *contextual attributes*, such as time, only partly and the generated signatures can *hardly be read or manually adapted*.

Hence, existing work isn't suitable to answer the following research questions:

RQ1 How can anomalies in textual process instance data be detected?
RQ2 How can contextual attributes be used to improve anomaly detection?
RQ3 How can signatures be automatically generated and described in a human readable and adaptable way?

Therefore, we propose a novel automatic signature generation approach which enables to exploit contextual attributes to improve anomaly detection. The signatures are defined as human readable regular expressions (regex). The applica-

[1] Note, those can be extracted from recorded process execution logs which are frequently automatically generated by process execution engines.

bility of the proposed approach is shown based on a proof-of-concept implementation which analyzes $240,000$ anomalous data entries.

This paper is organized as follows. The process instance data signature syntax and the integration of contextual attributes are discussed in Section 2. The proposed signature generation approach is defined in Section 3. Evaluation, corresponding results and their discussion are presented in Section 4. Section 5 discusses related work. Conclusions and future work is given in Section 6.

2 Signatures on Textual Process Instance Data

To protect process instances from unknown attacks we propose an automatic signature generation approach which enables anomaly detection during process execution. The signatures are generated from recorded process execution logs that are created automatically by process execution engines during runtime [14]. Such logs hold, for example, all variables—including their values—which are used by a process model during its execution [14, 5]. Hence, a signature can be created that matches the recorded variable data (e.g., variable values exchanged between process activities or received from external partners, such as, other processes/ web services) and therefore allows to distinguish between the recorded—expected and typical—behavior and anomalous behavior that would be observable, e.g., during attack preparations. Detecting such anomalies allows to apply various counter measures, such as stopping the execution of affected process instances.

We assume that especially processes and process execution engines are a *worthwhile application area* for anomaly detection, because, today's processes integrate and share the data of a wide range of services and applications. Hence, integrating signature generation and anomaly detection directly into process execution engines enables to secure a huge amount of potential attack areas at once and provides a direct access to all the required data.

We propose that the signatures should be created automatically to meet the complexity and flexibility of today's business processes [4]. Process repositories frequently contain hundreds of individual process models, which are executed in a versatile service landscape [12]. Hence, a large amount of signatures must be created and constantly updated. Additionally, we assume that the documentation of each service/application that is integrated in the processes is frequently outdated or missing because changes are often implemented in a rapid pace leaving less time to update a constantly increasing amount of documentation. This leads to high signature generation costs or incorrect signatures. To address this challenge we propose to extract signatures (i.e., expected behavior) from real process executions described in recorded process execution logs. Doing so ensures that the signatures match the real behavior and not, e.g., some incorrect behavior based on an outdated manual. Additionally, the log data contains all the information that is required to create context specific signatures, which simplifies the generated signatures and increases their anomaly detection.

Contextual Process Instance Data Behavior that is anomalous only in a specific context, but not otherwise, is termed *contextual anomaly* [3]. Taking *contextual attributes* into account enables the creation of more focused signatures and hereby increases the anomaly detection rate. So far, contextual attributes were neglected for anomaly detection in the business process domain.

Imagine that signatures should be created for data that is received by two process activities. The first activity is always receiving characters while the second one is always receiving digits. Without contextual attributes only a single signature would be created to check if only characters *or* digits are received. However, when the second activity surprisingly starts to receive characters, then this could not be detected because the signature only checks for characters *or* digits. But when taking contextual attributes into account—here the context is defined as the activity which receives the data, e.g., from an external service—two independent signatures are created for each activity and the behavioral change is detected as an anomaly. Hence, taking the context into account increases the performance (i.e., anomaly detection rate) of the generated signatures and also simplifies the signatures itself. Moreover, two specialized separate signatures—one to match digits and one to match characters—are shorter and easier to read and to maintain than a single signature that has to fulfill multiple tasks at once.

Two kinds of contextual attributes are exploited in the following:

General attributes such as time (by creating unique signatures for specific time periods, such as individual months), because, we assume that process instance data values can be time dependent.

Process instance specific attributes for example, which process or activity has created the analyzed instance data values because we assume that the data can greatly differ between different activities/processes and even between multiple activity data fields/variables.

Further kinds of contextual attributes, for example, which user has defined the observed variable values, are left for future work.

The signature generation starts with a pre-processing step that groups the analyzed data based on the discussed contexts to create an unique signature for each contextual attribute combination. In the following, an anomaly detection system can select the appropriate signature based on the observed context.

Signature Definition The signatures are defined as regular expressions, for the following reasons: Regular expressions are well-suited to analyze textual data, are supported by many programming languages, human readable and adaptable, and used by existing intrusion detection systems (such as Snort or I7-filter). Further on, high speed matching algorithms are available [15]. We assume that these advantages ease the integration of the presented anomaly detection approach into existing process execution engines and anomaly detection systems.

The signatures (i.e., regular expressions), defined in the following, consist of a choice operator, groups, and multiple metacharacters which are listed in Table 2. Note, that the '^' and '$' character get added to the start ('^')/end ('$') of each signature to enforce that the whole observed content matches the signature.

The regular expression syntax presented in Table 2 enables the definition of *simple* signatures which match the *structural components* of the observed data (e.g., for XML data this would be the XML tags). Additionally, we propose a

Table 2. Simple regular expression signature syntax.

Character	Description
‘^’	Matches the start of the compared data.
‘$’	Matches the end of the compared data.
‘.’	Matches any possible character (including digits and control characters).
‘{n}’	Matches the preceding expression, exactly *n* times.
‘{a,b}’	Matches the preceding expression, between *a* and *b* times.
‘|’	Matches either the expression before or after the vertical bar.
‘()’	*Capturing group* which concatenates expressions or groups of expressions. Note, a capturing group can be defined in combination with ‘{n}’, ‘{a,b}’, or ‘|’. For example, ‘(ab|cd)’ matches ‘ab’ or ‘bc’ while ‘(.){5}’ matches any possible character exactly five times.

Table 3. Extensions for the simple regular expression signature syntax.

Character	Description
‘\w’	A character class that matches any word character.
‘\s’	A character class that matches any formating character such as tabs or spaces.
‘\d’	A character class that matches any digit.
‘[^\s\d\w]’	A negated combination of multiple characters which matches any character that is not matched by ‘\w’,‘\s’, or ‘\d’, e.g., a minus sign (‘-’).

novel approach to increase the anomaly detection performance by taking the content of the observed data into account (e.g., for XML this would be the data which is placed between the XML tags, i.e., a XML node value). Accordingly, the presented syntax (cf. Table 2) is extended with character classes (cf. Table 3) to define *complex* signatures.

Character classes allow to differentiate if an observed character is, e.g., a number or a letter. Hence, it becomes possible, for example, to define a signature which checks if the observed bank account number always starts with two letters and ends with at least four numbers. This enables the generated signatures to ensure that structural and content-related properties comply with the expected behavior. An example for a signature that was defined using the described syntax can be found in the running example in Table 1.

3 Generating Signatures

This section presents a novel automatic process instance data signature generation approach. From a given set of training data (e.g., recorded process model instance executions) it generates signatures that allow to detect if the currently observed behavior (i.e., values of textual variables that are exchanged/used during ongoing process instance executions) are anomalous (i.e., do not match the expected common behavior specified at the signature) or not.

Each signature is generated by four main components (cf. Fig. 1). *(1)* the *pre-processing module* extracts the relevant data from process execution logs. Subsequently, the textual variables—strings—are grouped based on various contextual attributes and each group is individually forwarded to the tokenization module. *(2)* the *tokenization module* identifies tokens (i.e., substrings) which commonly occur in the recorded variables. Next, the position and order of the tokens in the analyzed data is extracted and used to group and combine the tokens. *(3)* is a module that analyzes the text that is placed *between tokens*. The *simple signatures* only utilize the length of the text that is placed between the tokens. Alternatively, *complex signatures* are constructed by converting the text

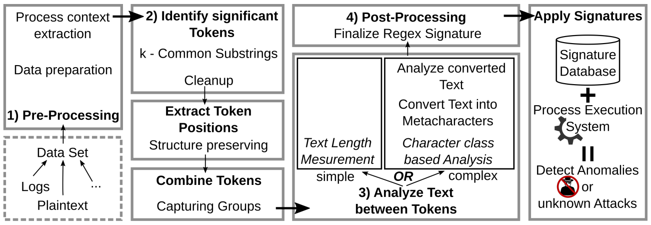


Fig. 1. Abstract signature generation approach overview.

between the tokens into character classes and injecting the result into module *(2)* to check if the converted text reveals previously hidden structures. *(4)* the *post-processing module* takes all the extracted and prepared information and constructs a valid regular expression which can be stored in a signature database and used to detect anomalous behavior during process model instance executions.

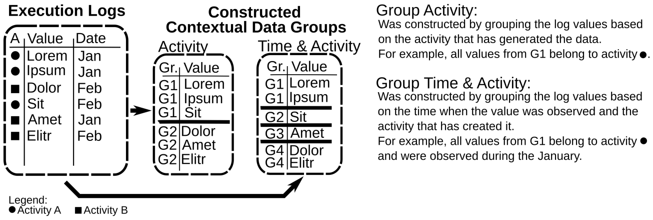
Pre-Processing The purpose of the pre-processing step is twofold: First, recorded process execution logs are prepared so that their content can be processed by the following steps. For this, the logs are analyzed and all variables with textual information and their metadata (i.e., important contextual attributes, e.g., which activity a variable belongs to or when it was created) are extracted.

Secondly, the prepared data is *grouped* based on the *identified contextual attributes* (e.g., time, activity, or process model, hence, the same variable value can be contained in multiple groups). For example, all textual data that belongs to one specific variable in one activity/process model is added to the same group and processed at once. Hereby, individual signatures for each contextual attribute (e.g., time or activity) and their combinations are created. Hence, contextual differences are respected during signature generation to ensure that a variable's content behaves as expected in specific situations, for example, in a specific month, process, or activity. For example, we found—while evaluating this paper—that processes contain activities which always assign the same values to their variables depending on the month when the activity is executed (e.g., each January), while the same activity variables contain a diverse set of values when comparing different months (e.g., January vs. June). Hence, creating an independent signature for each contextual attribute (e.g., time of the year) increases the signature anomaly detection performance, because more fine granular signatures are generated which focus on specific situations and contexts. Additionally, the signatures become simpler (e.g., because only a subset of the available data must be covered by a single signature) and therefore easier to read and maintain.

An example for creating contextual groups is depicted in Fig. 2. For the sake of brevity the execution log data only contains a single process with two activities. Two contextual groups are constructed—based on activity only and the combination of activity and time.

The following steps, starting with identifying significant tokens, are then applied on the textual variable data stored in each generated contextual group.

Identifying Significant Tokens *Significant tokens* are substrings which commonly occur in the recorded and analyzed process instance textual (string) variable values. Those tokens are used to construct signatures that detect anomalies



A	Value	Date
●	Lorem	Jan
●	Ipsum	Jan
■	Dolor	Feb
●	Sit	Feb
■	Amet	Jan
■	Elitr	Feb

Gr.	Value
G1	Lorem
G1	Ipsum
G1	Sit
G2	Dolor
G2	Amet
G2	Elitr

Gr.	Value
G1	Lorem
G1	Ipsum
G2	Sit
G3	Amet
G4	Dolor
G4	Elitr

Group Activity:
Was constructed by grouping the log values based on the activity that has generated the data.
For example, all values from G1 belong to activity ●.

Group Time & Activity:
Was constructed by grouping the log values based on the time when the value was observed and the activity that has created it.
For example, all values from G1 belong to activity ● and were observed during the January.

Fig. 2. Exemplary contextual group construction.

(i.e., detect that a significant token that is present at all the analyzed data is surprisingly missing). The problem of finding the significant tokens is defined as:

INPUT: A list of strings to analyze S, a minimum token length of $tl_{min} \in \mathbb{N}$ characters, and a minimum occurrence of $to_{min} \in (0,1]$, e.g., the token must occur in at least 10 percent of the analyzed strings in S to be significant.

OUTPUT: A list of distinct substrings D and therefore significant tokens which fulfill the minimum length (tl_{min}) and occurrence requirement (to_{min}).

The *k-common substring algorithm* [6] is applied to this problem to identify the longest substrings which occur in at least k (i.e., to_{min}) strings. For this, a *generalized suffix tree* [6] is generated from S. We have extended the suffix tree data structure so that each tree node contains the list of strings in S that are represented by this node (cf. to_{min}) and the respective length of each sub path (cf. tl_{min}). This enables to ensure the compliance with to_{min} and tl_{min} when extracting significant tokens. The extraction itself starts from *terminator nodes* (i.e., tree nodes that hold the last character/substring of a string in S). For each terminator node that fulfills tl_{min} the algorithm traverses towards the root of the tree until it finds a node that fulfills to_{min}. Then it starts recording the data that is stored in each node until it reaches the root node and therefore has identified a *potentially significant token* which is then stored in D. This is repeated for each terminator node to extract all potentially significant tokens.

Finally, all the potentially significant tokens are cleaned up. First, duplicates are removed so that each potentially significant token only appears once in D. Secondly, each remaining potentially significant token in D is analyzed to identify if it can be completely replaced by one of the longer[2] tokens in D. Why? Because a longer token provides a more strict representation of the analyzed data because it enforces more characters. Imagine that S, inter alia, contains the words '*`performance`*' and '*`performed`*' and that, among others, '*`perform`*' and '*`for`*' are commonly occurring tokens hold by D. S and D are now evaluated by checking, for each token and analyzed string, if a shorter token could be completely replaced by a longer token. Hence, for the analyzed words '*`performance`*' and '*`performed`*' it is checked if for all positions were the token '*`for`*' occurs also the token '*`perform`*' occurs. This is the case and so '*`for`*' will be removed from D to replace it with the longer token '*`perform`*'.

Applying the described token identification approach to the running example (cf. Table 1) results in following significant tokens (when defining tl_{min} as 5 and to_{min} as 0.2): '*`a=mapred:`*','*`␣PCMA/8000`*','*`␣GSM/8000`*', and '*`/8000`*'

[2] Measured based on the number of characters.

Extract Token Positions The position, order, and occurrence of each significant token is determined for each string in S. Hereby, tokens that are placed on related positions are identified. Subsequently, these tokens are used to form regex groups. The problem of finding token positions/order is defined as:

INPUT: A list of strings S and a list of cleaned up significant tokens D.

OUTPUT: A list P were each $p \in P$ is a list of significant tokens which occur in the respective $s \in S$ ordered based on their position in s.

For each $s \in S$ the left most positioned token[3] $d \in D$ is identified. If such a token was found then it is stored in p, and s is trimmed to remove all characters left from the position where d ends. Subsequently, the search for the left most significant token restarts on the trimmed version of s. This repeats until no more significant tokens can be found in the trimmed s. Note, for each $s \in S$ an respective $p \in P$ is created and utilized/filled.

The following P is generated for the running example's tokens and strings (cf. Table 1) (the list entries are separated using semicolons for P and commas for p). This allows to deduce, for example, that '*a=mapred:*' is present in all $s \in S$ (i.e., all analyzed strings) and that it is always the left most significant token: '*a=mapred:,␣PCMA/8000*'; '*a=mapred:,␣GSM/8000*'; '*a=mapred:,/8000*'; '*a=mapred:,␣GSM/8000*'; '*a=mapred:,␣GSM/8000*'

The ordered tokens and their positions are used during the next step to start with the creation of regular expressions (i.e., signatures).

Grouping Tokens based on their Order We propose that the generated signatures should represent the *structural components* (represented by significant tokens) of the analyzed data (e.g., for XML data this would, likely, be the XML tags). However, we assume that most likely not each analyzed string will contain the exact same significant tokens. Hence, *regex groups* are created to enable the signature to choose from multiple token *alternatives*, for example to specify that token A or B should occur. Additionally, we expect that the analyzed textual data is of *variable length* so that the structural components are most likely not overlapping (i.e., use the same absolute positions) for each string in S. Hence, it is not possible to decide which tokens should be grouped solely based on the absolute position of the tokens. Accordingly, we propose to group the identified significant tokens based on the order of their occurrences rather than on their absolute positions. The problem of grouping the tokens is therefore defined as:

INPUT: P, as defined in the previous step.

OUTPUT: A list G were each $g \in G$—for each $p \in P$ an associated $g \in G$ is generated—holds a list of significant tokens that are combined into regex groups.

To combine the tokens the algorithm identifies the shortest entry $p \in P$ (i.e., it is containing the least amount of significant tokens) and extracts its length as $y \in \mathbb{N}$. y is then used as the amount of tokens which should be grouped. To group the tokens a list of indexes ranging from—if y is even—0 to $(\lfloor y/2 \rfloor - 1)$ is created. Subsequently, from each token list $p \in P$ the tokens with the respective indexes are taken and stored in a new list $g \in G$ (first to last, an independent list

[3] If two tokens start on the same position then the longer one is chosen because it enforces more characters during signature checking than a shorter one.

g is generated for each p). A similar approach is applied on the second half of the indexes (i.e., $(|p| - \lfloor y/2 \rfloor) \cdots (|p| - 1)$). However, this time the algorithm iterates from the last token in each $p \in P$ towards the first token (last to first) and adds the tokens (in reversed order) to the already existing $g \in G$ that belongs to the respective p. If y is uneven then an additional iteration is executed to cover the token index which would else be ignored (i.e., $0 \cdots \lfloor y/2 \rfloor$ is used at first to last).

The approach described above ensures that the generated signatures cover a wide area of the analyzed data. Imagine, that the approach would only incorporate a single direction (e.g., first to last) then an attacker could attach the vulnerable information to the end of the data—especially if the amount of tokens in each p fluctuates. Secondly, we found a positive impact of this two direction approach during the preliminary evaluation, especially, when analyzing XML data because the two direction approach more frequently preserved matching XML start/end tags and therefore more likely recognized missing XML nodes.

Finally from each $g \in G$ the tokens with equal indexes (e.g., all first tokens, all second tokens, and so on) are combined into distinct *regex groups* using the *or* operator (‘|’). For the running example (cf. Table 1) the following regex groups are generated: ‘`(a=mapred:)`’; ‘`(␣PCMA/8000|␣GSM/8000|/8000)`’

The significant tokens likely do not represent all the analyzed data, for example, data which is not occurring frequently enough to become a significant token (e.g., varying content that is placed between XML tags). Hence, a novel approach to integrate the remaining data into the generated signatures is presented.

Analyze Textual Data between Tokens Until now the textual data which is placed between the identified significant tokens was not yet addressed. This data mainly consists of application data, such as addresses or names, which frequently do not contain stable structural components. However, this data is processed by the process activities and should, therefore, also be checked for anomalies to prevent attackers from injecting vulnerable—anomalous—data. Hence we propose two novel approaches called *simple* and *complex*.

INPUT: A list S and a list G, as defined in the previous step.

OUTPUT: Regex artifacts that represent the textual data between the tokens. Hence, the simple approach utilizes the length of the respective strings between the tokens to represent them. For the complex approach the representations are generated from a mixture of length information and character classes.

Both, the complex and the simple approach, analyze the textual data that is positioned between the identified significant tokens (e.g., this is, for XML data, likely, the data between XML tags). So, this data must first be extracted. Therefore, for each string $s \in S$ the respective list of significant grouped tokens $g \in G$ that occur in s is exploited. Hence, g is used to identify the position of each significant grouped token in s. Further on, the text between each identified token position and its predecessor token is extracted and stored for future analysis. A similar approach is used to extract the text between the first/last token and the start/end of s. Hence, all text that is placed, for example, between the second and the third token (for each $g \in G$) is, in the following, processed at once.

For the running example (cf. Table 1) the following strings are identified as text that is placed between the two generated groups of significant tokens: ‘`8␣set`’; ‘`3␣startFAPCM`’; ‘`0␣test␣PCMU`’; ‘`3␣hello`’; ‘`5␣stop`’. Subsequently these strings are processed by a complex or a simple approach.

Complex: The *complex approach* converts the textual data into a format that makes it more likely to identify structural information. Imagine, that some bank account numbers should be analyzed (e.g., ‘`AB12345`’, ‘`GH56521`’, and ‘`UJ56122`’). Initially the token based analysis is not able to detect significant tokens and therefore structure, because, each bank account number is unique and substrings which occur at multiple account numbers can, therefore, not be identified. However, a close analysis reveals that each account number starts with two letters, continued by five digits. To enable the presented complex approach to recognize this pattern the data is converted in an *abstract representation.*

Therefore, each letter is converted into a ‘`w`’, each formating character (e.g., a space) into a ‘`s`’, each digit into a ‘`d`’, and any other character is converted into an ‘`r`’. Hence, each account number is then represented as ‘`wwddddd`’. Subsequently, the presented signature generation approach is applied on the prepared data (three times ‘`wwddddd`’, one for each abstracted account number), starting from the “Identifying Significant Tokens” step. Hereby the regex group ‘`(wwddddd)`’ is generated to represent the fact that each analyzed string contains two letters and five digits. Finally, the characters (‘`w`’,‘`s`’,‘`d`’, and ‘`r`’) are replaced with regular expression character classes (‘`w`’ → ‘`\w`’, ‘`d`’ → ‘`\d`’, ‘`s`’ → ‘`\s`’, ‘`r`’ → ‘`[^\s\d\w]`’), cf. Table 3, which enforce, during signature checking, the specified order and occurrence of digits, letters, formating characters, and so on. Hence, ‘`(wwddddd)`’ becomes ‘`(\w\w\d\d\d\d\d)`’. Note, that the complex approach falls back to the simple approach for parts of data where no structure (even when applying the discussed abstraction approach) could be identified.

Simple: The *simple approach* deals with the textual data in a more abstract way than the complex one. Hence, it analyzes the respective data and identifies the shortest and the longest string. Subsequently, the length of these two strings is used to add minimum/maximum length limits to the signatures. Hence, when applying it on the running example (cf. Table 1) the following signature artifacts are generated: The shortest identified textual information is ‘`8␣set`’ and the longest is ‘`3␣startFAPCM`’. So the content is described as ‘`(.){4,13}`’ which indicates that any possible text is valid, but, it must be between 4 to 13 characters long. A shorter definition is used if each string is of equal length, for example, ‘`(.){4}`’ if each analyzed string is exactly 4 characters long.

For the running example (cf. Table 1) the complex approach generates the following regex artifact: First, the substrings which are placed between the two identified token groups are abstracted: ‘`8␣set`’ → ‘`dswww`’, ‘`3␣startFAPCM`’ → ‘`dswwwwwwwwww`’, ‘`0␣test␣PCMU`’ → ‘`dswwwwswwww`’, ‘`3␣hello`’ → ‘`dswwwww`’, ‘`5␣stop`’ → ‘`dswwww`’. Then, the complex approach identifies ‘`dswww`’ as a structural component (i.e., significant token). Why not use ‘`dswwwwwww`’? Because ‘`dswww`’ is the only substring that fulfills the minimum length requirement and occurs frequently enough in S (when using $tl_{min} = 5$ and $to_{min} = 0.2$). However, ‘`dswww`’

is not able to represent all the strings (e.g., ‘`dswwwwwww`’ contains more characters than ‘`dswww`’). Hence, also the simple approach is applied as a fall back. Altogether, the following result is generated to represent the data which is placed between the two identified significant token groups: ‘`(\d\s\w\w\w)(.){0,7}`’
Post-Processing All the components (e.g., token groups) are now combined to create a signature that is a valid regular expression. Subsequently, the signature can be stored in a signature database and used by process execution engines to detect anomalies and, therefore, potential attacks or attack preparations.

During post-processing three objectives are fulfilled. First, all generated components (e.g., the token groups) are combined to generate a *raw signature*. It is called raw signature because it is not yet ready to be stored in a signature database. Secondly, the characters which have a special meaning in regular expressions are, if necessary, escaped. For example, the plus sign (‘`+`’) typically indicates that some character should be matched at least once. However, if a plus sign should be treated as a normal character (e.g., because it is a part of a significant token) it must be escaped by placing a backslash (‘\’) in front of it. Thirdly, a circumflex (‘`^`’) is placed at the start of the signature and a dollar sign (‘`$`’) is placed at the end. Why? Because this enforces that the signature must match the whole observed data from the start to the end and not only a part of it. Hence, it increases the anomaly detection performance of the generated signatures because an attacker can no longer send some valid data and then attach the vulnerable data to the end of it, which would otherwise be possible.

For the running example (cf. Table 1) the finalized signature is defined as: ‘`^(a=mapred:)(\d\s\w\w\w)(.){0,7}(␣PCMA\/8000|␣GSM\/8000|\/8000)$`’

4 Evaluation

The evaluation combines *realistic artificial data* and *real life process instance execution data* to assess the impact of contextual attributes on signature generation and the anomaly detection performance of the presented approach.
Test problems The test data which was used for the evaluation consists of a*) artificially generated test data*[4] (using three different formats, namely, XML, JSON, and EDIFACT) and *b) real life process execution logs* from the Business Processing Intelligence Challenge 2015[5] (provided by five Dutch municipalities).

For a*)* the artificially generated data consists of three different data formats (XML, JSON, and EDIFACT—wide spread data exchange formats, used in various disciples, such as banking or manufacturing) that we found are frequently used in business processes. For each of the three formats a thousand test data entries were generated and randomly separated into *signature generation data* and *test data*. Three hundred (100 for each data format) randomly selected entries from the test data were also used to construct anomalous data to evaluate the anomaly detection performance of the presented approach. Moreover, the generated XML, JSON, and EDIFACT data entries contain realistic data as payload

[4] `http://cs.univie.ac.at/wst/research/projects/project/infproj/1057/`

[5] `http://www.win.tue.nl/bpi/2015/challenge`

(e.g., realistic e-mail addresses, phone numbers, or company names) along with the required structural components (e.g., XML tags). Each generated XML and JSON data entry holds 4 payload values (e.g., names), while each generated EDIFACT data entry represents a purchase order message with 14 payload values.

For *b)* the realistic log data consists of $262,628$ independent events from 27 process models and 356 activities—recorded over a period of six years.

Real life and artificial data were combined because the identified real life data only contains simple textual variable values (i.e., textual variables typed as strings [5] that hold dates, booleans, or numbers) which can easily be addressed by the presented approach. Hence, we opted to include complex artificially generated data to assess the performance of the presented anomaly detection approach in situations where the data is complex and, therefore, more challenging. Note, despite the prototypical implementation, the signature generation could be conducted quickly (5 min to generate signatures for all test data items, fractions of a section to decide if a value is anomalous or not – on a 2.6Ghz Intel Q6700).

Metrics and Evaluation *Quantitative* and *qualitative metrics* were combined.

Quantitative: Realistic artificially generated data (i.e., signature generation data) was used to generate signatures, one for each data format. Subsequently, each signature had to match the respective test data to ensure that the signature was not *over-fitted* [7]. An over-fitted signature can lead to many false positives which would reduce the applicability of the presented approach. Note, each generated signature successfully evaluated the test data as non-anomalous. So, no over-fitting occurred. Finally, each signature was applied to anomalies that were generated from the test data to assess its anomaly detection performance.

Qualitative: Real life process execution logs were analyzed to check if contextual attributes, such as time, have an effect on the variables and data fields, used by the process, that would allow to improve anomaly detection. For example, it was evaluated if the variable values of an activity show similarities for specific times of the year (e.g., each April, for multiple years). If this is the case, then respecting contextual attributes (e.g., time) and therefore creating an independent signature for each month is beneficial because less data must be represented by each single signature which improves the anomaly detection performance.

Results The results were generated by applying the signatures on randomly selected test data entries which were altered to represent 8 *anomaly classes*.

The following anomaly classes were evaluated: *a)* The length of the data entry was extended by 4-10 random characters, *b)* The data entry was completely replaced by random characters, *c)* Content (e.g., for XML data this is the value of a XML node) was replaced by random characters, *d)* Content was duplicated and attached to the original value, *e)* Between 4-10 characters of the content were randomly selected and flipped (e.g., a letter was replaced with a random digit), *f)* An element (e.g., a complete XML node) of the data entry was completely removed, *g)* An element (e.g., a complete XML node) of the data entry was duplicated, *h)* A structural element (e.g., a XML tag) was replaced with random data. Note, that the anomaly classes *b)*, *c)*, and *h)* replaced data with a randomly generated equivalent that has the exact same length as the replaced data.

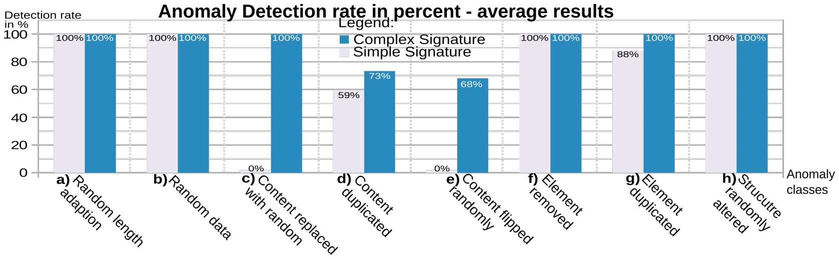


Fig. 3. Anomaly detection performance of simple and complex signatures.

We assume that the generated anomalous data entries realistically represent data that can be observed during attacks. For example, the anomaly class *d)* can be used to check for potential buffer overflows or the anomaly *c)* represents the attempt to inject machine code into a process model instance. Overall 240,000 anomalous data entries were generated and evaluated. The evaluation was executed a hundred times to even out the random behavior of the anomaly class adaptation approach. During each execution a hundred test data entries were individually adapted by 8 anomaly classes, for three different data formats.

Primary tests were executed to identify appropriate configuration values for the discussed signature generation approach, resulting in $tl_{min} = 4$ and $to_{min} = 0.75$. The average results of the evaluation are shown in Fig. 3.

The results show that the presented approach is capable of detecting a wide range of anomalies. Already the simple approach generates reasonable results for most anomaly classes. However, the simple approach is not able to detect anomalies that only affect the content (e.g., XML node values) of the analyzed data without changing its length (e.g., only specific characters are replaced, cf. anomaly *c)* and *e)*). This is not surprising because the simple approach only enforces length restrictions on the content. Here, the complex approach comes into play. By analyzing the content and its internal structure it can, for example detect flipped characters (e.g., anomaly *e)*). Hence, we conclude that the presented novel complex signature generation approach is capable of providing remarkable strict signatures while the simple signatures are easier to read. They are shorter, and are already able to detect important length based vulnerabilities (e.g., buffer overflows). Why are anomalies *d)* and *e)* not always detected? This can occur, for *d)*, if the duplicated value is still shorter then other representations of this value in the signature generation data or, for *e)*, if the flipped character value is also present at the same place at data entries in the signature generation data.

Table 4. Influence assessment of respecting contextual attributes.

No.	Context. Attribute	Beneficial	No.	Context. Attribute	Beneficial
1	Process activity	Yes	**3**	Time	Yes
2	Process model	Yes	**4**	Combination	Yes

The importance of contextual attributes for process signature generation was evaluated using process execution logs provided by the Business Processing Intelligence Challenge 2015. It was checked, for three different contextual attributes (activity, process, time, along with their combinations), if the generated signatures *benefit* from respecting these contextual attributes during signature generation (e.g., by generating an independent signature for each activity and month). We found clear indications that the recorded data is influenced by the described attributes, cf. Table 4. For example, some activities always used the same variable values during specific times of the year or when integrated into specific process models. Moreover, we found that activities, despite equal variable names, store vastly different data formats. We conclude that respecting contextual attributes during the signature generation allows to generate simpler signatures and increases the signature anomaly detection performance (because less diverse data must be covered by a single signature, so the signature becomes easier to read/ maintain and it can be more strictly represent the analyzed data).

5 Related Work

Related work, in the business process anomaly detection domain, can be classified into two categories: process instance data and process model control flow anomaly detection. The existing *data anomaly* detection approaches concentrate on integer variables and apply statistical regression analysis to identify outliers and, therefore, anomalies [11]. *Control flow anomaly* detection approaches mine process logs to extract control flows which are then, for example, compared with a reference process model. Alternative approaches check how frequently each control flow is found, infrequent flows are then marked as anomalies, cf. [2, 1].

We conclude that textual business process instance data is currently not addressed by existing process anomaly detection approaches. Moreover, we found that contextual attributes are currently not exploited in the business process domain. In general, in the *security domain*, anomaly detection in textual data is currently mainly applied to detect novel topics in a collection of documents [3] or on highly standardized network protocols [16], such as SIP, neglecting the security critical aspects of arbitrary textual data. This circumstances reduce the protection gained from today's, process, anomaly detection solutions.

6 Conclusion

This paper provides process instance anomaly detection and signature generation approaches ($\mapsto$ **RQ1** to **RQ3**) which will be integrated in our "ProTest" project which focuses on creating automatic process behavior verification. Future work will exploit the generated signatures as a foundation to construct realistic test data to improve process model testing. In addition, we are confident that the described approach can also be applied to related domains (e.g., web services) that process textual data and, even, other data types (e.g., binary data).

The evaluation results show the flexibility and applicability of the presented approach for complex data formats ($\mapsto$ **RQ1**). Additionally, we found that contextual attributes affect the analyzed business process instance data and conclude that contextual attributes can be exploited to improve the signature quality (i.e., anomaly detection performance; ↦ **RQ2**). Overall, this work provides the first process instance anomaly detection approach that addresses textual data and enables to replace error prone manual signature generation (↦ **RQ3**).

Future work will strive to enhance the performance of the generated signatures, and to identify ways which enable to measure how much the observed behavior deviates from the expected one. Hereby multiple anomalies and their effects can be aggregated to decrease the risk of improperly assessing small, probably harmless, anomalies, as large, probably harmful, anomalies (i.e., attacks).

Acknowledgment The research was partly funded by COMET K1, FFG - Austrian Research Promotion Agency.

References

1. Bezerra, F., Wainer, J.: Algorithms for anomaly detection of traces in logs of process aware information systems. Information Systems 38, 33–44 (2013)
2. Bezerra, F., Wainer, J., van der Aalst, W.M.: Anomaly detection using process mining. In: Enterprise, Business-Process and Information Systems Modeling, pp. 149–161. Springer (2009)
3. Chandola, V., Banerjee, A., Kumar, V.: Anomaly detection: A survey. ACM computing surveys 41, 15–87 (2009)
4. Fdhila, W., Rinderle-Ma, S., Indiono, C.: Change propagation analysis and prediction in process choreographies. Cooperative Information Systems 24, 47–62 (2015)
5. Günther, W.C., Verbeek, E.: XES – Standard. Tech. rep., TU Eindhoven (2014)
6. Gusfield, D.: Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press (1997)
7. Hawkins, D.M.: The problem of overfitting. Chemical Information and Computer Sciences 44, 1–12 (2004)
8. Herrmann, P., Herrmann, G.: Security requirement analysis of business processes. Electronic Commerce Research 6, 305–335 (2006)
9. Liao, H.J., Lin, C.H.R., Lin, Y.C., Tung, K.Y.: Intrusion detection system: A comprehensive review. Network and Computer Applications 36, 16–24 (2013)
10. Müller, G., Accorsi, R.: Why are business processes not secure? In: Number Theory and Cryptography, pp. 240–254. Springer (2013)
11. Quan, L., Tian, G.s.: Outlier detection of business process based on support vector data description. In: Computing, Communication, Control, and Management. pp. 571–574. IEEE (2009)
12. Rosemann, M.: Potential pitfalls of process modeling: part b. Business Process Management 12, 377–384 (2006)
13. Sneed, H.M.: Integrating legacy software into a service oriented architecture. In: Software Maintenance and Reengineering. pp. 11–22. IEEE (2006)
14. Van Der Aalst, W.: Process mining: discovery, conformance and enhancement of business processes. Springer (2011)
15. Yamagaki, N., Sidhu, R., Kamiya, S.: High-speed regular expression matching engine using multi-character nfa. In: Field Programmable Logic and Applications. pp. 131–136. IEEE (2008)
16. Zuech, R., Khoshgoftaar, T.M., Wald, R.: Intrusion detection and big heterogeneous data: a survey. Big Data 2, 1–41 (2015)