Comparing Approaches to Mining Source Code for Call-Usage Patterns

Huzefa Kagdi[1], Michael L. Collard[2], and Jonathan I. Maletic[1]
[1]Department of Computer Science
Kent State University
Kent Ohio 44242
[2]Dept of Mathematics & Computer Science
Ashland University
Ashland Ohio 44805
{hkagdi, collard, jmaletic}@cs.kent.edu

Abstract

Two approaches for mining function-call usage patterns from source code are compared. The first approach, itemset mining, has recently been applied to this problem. The other approach, sequential-pattern mining, has not been previously applied to this problem. Here, a call-usage pattern is a composition of function calls that occur in a function definition. Both approaches look for frequently occurring patterns that represent standard usage of functions and identify possible errors. Itemset mining produces unordered patterns, i.e., sets of function calls, whereas, sequential-pattern mining produces partially ordered patterns, i.e., sequences of function calls. The trade-off between the additional ordering context given by sequential-pattern mining and the efficiency of itemset mining is investigated. The two approaches are applied to the Linux kernel v2.6.14 and results show that mining ordered patterns is worth the additional cost.

1. Introduction

A function-call-usage pattern is a list or set of function calls found in the source code. Although many of these call-usage patterns are very intuitive and may be common knowledge to developers, they are typically not documented. They form latent programming rules that seldom exist outside the minds of developers. Violations of these types of rules are difficult to uncover, report to bug-tracking systems, and fix unless the rules are explicitly documented and made available.

Recently, researchers [7, 8, 10] have applied data-mining techniques, specifically frequent-pattern mining algorithms, to the problem of uncovering/discovering call-usage patterns from the source code of large systems. The result is a set of rules that describe frequently occurring call-usage patterns within a system. This has been found to be potentially useful for tasks such as identification of standard library/API usages and fault location [7, 8, 10]. The techniques used are quite efficient; however they sometimes result in a large number of false positives, i.e., reporting potential errors where none actually occur. Large numbers of false positives tend to alienate the users of such tools. But to more accurately identify faults one must apply fairly complex, and computationally expensive, static/dynamic analysis techniques [8]. On a large system this is not feasible. A data mining or similar approach can be used to reduce the space to a more reasonable size so that more sophisticated static/dynamic approaches can realistically be applied.

To date, the work using data mining has only used one basic technique, namely *itemset mining*. While this is a very efficient technique, it is not always very accurate and often produces many false positives [7]. This is, in part due to the fact that itemset mining produces patterns that are unordered sets of function calls. So an itemset pattern about the function calls *open* and *close* would allow any ordering of these two calls (e.g., *open(); ... close();* or *close(); ... open();*).

To improve on the itemset-mining approach another frequent-pattern mining technique, namely *sequential-pattern mining*, is applied to this problem. Sequential-pattern mining results in a partially ordered list of function calls for the usage pattern. Depending on the domain, the results of sequential-pattern mining tend to be more accurate and have fewer false positives than itemset-mining results. However, sequential-pattern mining comes with a higher computational cost. Here we compare the results of both techniques to ascertain if sequential-pattern mining is a better method to be used for call-usage pattern mining. The comparison is based on accuracy of patterns, candidate patterns and their violations, and computational costs for the version 2.6.14 of Linux kernel. The first issue is very difficult to definitively validate and we use a combination of manual inspection and examination of later versions of the same software system.

The paper is organized as follows. The two techniques are described and the problem is described in Section 2. The comparison of the two techniques along with validation of the results is given in Section 3. We discuss related work in Section 4. Finally, our conclusions and future directions are presented.

2. Mining Call-Usage Patterns

A *call-usage pattern* is a set or list of function calls found in a segment of code. In this paper, we focus on the call-usage patterns in the function definitions of procedural languages, more specifically *C*. The goal of the mining process is to uncover call-usage patterns that occur frequently in a software system. The fundamental premise is that frequently-occurring patterns of function calls in a system reflect candidates for standard usages of a library or API. Additionally, if these standard patterns can be automatically reverse engineered then violations of these standard patterns can easily be identified.

To identify violations we must identify variations of frequently-occurring patterns. Specifically, a *variant* is a proper subset of a frequent pattern that occurs by itself in the systems but in far fewer numbers (e.g., one or two times). These variants are special cases, or possible errors or misusages. A pair of a variant and a function in which that variant occurs is referred to as a *violation*.

Mining call-usage patterns from source code can be considered as an instance of the general problem of frequent-pattern mining from any type of data [1]. Before we describe the two approaches, data-mining terminology that is relevant to the discussion is introduced. The input data to frequent-pattern mining algorithms are in the form of *transactions* (e.g., customer baskets or items checked-out together in market-basket analysis). Here, an individual transaction corresponds to a single function definition.

The *support* of a pattern is the number of transactions in which it occurs i.e., the number of functions in which it appears. A *frequent pattern* has a support at or above that of a user-specified *minimum support* in the considered dataset. Such frequent patterns are typically used to form association rules between a pair of patterns (e.g., when pattern *A* occurs pattern *B* also occurs). The *confidence* of a rule is used to determine the strength of an association rule, and is generally computed from the support of the two patterns to a value in the range [0, 1.0]. A high confidence for a rule means the two patterns that make up the rule co-occur in most transactions.

Itemset mining produces patterns that are unordered sets of function calls and we term these *unordered patterns*. Likewise, sequential-pattern mining produces patterns that are partially ordered lists of function calls so we term these *ordered patterns*. We now discuss the itemset and sequence mining techniques in more detail.

2.1. Itemset Mining

Itemset mining takes a given set of transactions that are composed of some items and finds all the frequently-occurring subsets of items that have at least a user-specified minimum support [5]. Itemset-mining techniques are used in a variety of domains. The most famous application is market-basket analysis for uncovering buying patterns of items frequently purchased together (e.g., beer and diapers frequently bought together).

Itemset mining performed with a specified minimum support produces a set of candidate unordered patterns from a system. Once such unordered patterns are uncovered, association rules can be generated from them to uncover candidate variants. An association rule is formed from a pair of unordered patterns such that the pattern obtained by their union is also a candidate pattern. The hypothesis is that an association rule with a very high confidence, but not the highest value of 1.0, is likely to contain a variant. Such an association rule indicates that it has a pattern that occurs by itself in very few transactions.

Itemset-mining approaches have been previously applied for mining call-usage patterns [7, 8]. An approach based on itemset and association-rule mining is taken by Li et al [7] for detecting common programming rules and their variants. They have shown one application of variants in locating potential bugs in a software system.

2.2. Sequential-Pattern Mining

Sequential-pattern mining takes a given set of sequences that are composed of items and finds all the frequently occurring subsequences that have at least a user-specified minimum support [9]. Sequential-pattern mining techniques are typically applied to datasets with temporal or other ordering information. For example, in analyzing market-basket data with the additional timestamp information, patterns such as customers who bought a *camera* are also likely to buy *additional memory* in the next month.

Since function-call usages are inherently ordered, another possible approach is to uncover call-usage patterns with the additional ordering information in the set of calls. Sequential-pattern mining produces a set of candidate ordered patterns with a specified minimum support. Here, the order of calls is determined by their lexical position in the function definition.

However, only partial ordering can be given to calls for which the considered language does not specify a standard order of call evaluations. Both *K&R* (Kernighan and Ritchie) and *ANSI/ISO C* standards leave an inherent order ambiguity of the evaluation of operands and non-deterministic order of function-call argument evaluation. For example, the order of function calls *a* and *b* in the expression *a()+b()* and in the function argument list *(a(), b())* of the call *f(a(), b())* is undetermined. Therefore, compilers take liberty in ordering the calls *a* and *b* and different compilers assign different ordering. For example, the compiler *gnu gcc* assigns the order *b*, *a* (right to left) and the compiler *hp aCC* assigns the order *a, b* (left to right).

The approach used here is based on the semantics according to the language standards. Therefore, calls involved in constructs such as expression and argument list in situations where there is non-determinism produce a partial ordering. In the example shown in Figure 1, functions *f2* and *f3* have the same partially ordered pattern $\{a, c\}\rightarrow\{b\}$ due to non-determinism in the occurrence of calls *a* and *c* in the expression *c()+a()*. Functions *f1* and *f4* form completely-ordered patterns. Partially-ordered patterns are quite different from the unordered usage patterns produced by itemset mining. The partially-ordered parts in ordered patterns are non-determinism cases, whereas unordered patterns ignore the ordering information even if it is deterministic.

Once these ordered patterns are uncovered, sequence rules can be generated to uncover variants. A sequence rule is formed between a pair of ordered patterns such that one of them is a (order-preserving) subset of the other. Similar to association rules, a sequence rule with a very high confidence is likely to contain a variant.

2.3. Examples

We first demonstrate the unordered and ordered pattern mining with the help of a synthetic example. Then we give specific examples uncovered from the Linux kernel (v2.6.14). Consider a hypothetical system with four functions as shown in Figure 1. We will use a minimum support of two for a candidate pattern, i.e., at least two functions must contain the pattern, and a minimum confidence of 0.65 for a variant, i.e., at most 35% of the functions that contain the pattern variant.

Though the functions in the example are incomplete and give very little context, it can be seen that they have similar implementations. We first discuss the unordered patterns and variants produced by itemset mining. Two patterns $\{a, b\}$ and $\{a, b, c\}$ are produced as candidates. The pattern $\{a, b\}$ has a support of four as the calls *a* and *b* occur together in all four functions, *f1*, *f2*, *f3*, and *f4*. Therefore, the association rule $\{a, b\}\Rightarrow\{c\}$ can be formed from these two patterns with a confidence of 0.75 (i.e., three of the four functions that contain calls *a* and *b*, also contain a call to *c*). As this is greater than the specified minimum confidence, the pattern $\{a, b\}$ is reported as a candidate variant. The missing call *c* that makes the pattern {*a*, *b*} a variant is only absent in function *f1*. Therefore the function *f1* is reported as a function with the variant $\{a, b\}$ of pattern $\{a, b, c\}$ which causes the pair ($\{a, b\}$, *f1*) to be a candidate violation.

In case of sequential-pattern mining, three ordered patterns $\{a\}\rightarrow\{b\}$, $\{c\}\rightarrow\{b\}$, and $\{a, c\}\rightarrow\{b\}$ are reported as candidates. Pattern $\{a\}\rightarrow\{b\}$ occurs in functions *f1* and *f2* where call *a* is an argument to call *b*, and also occurs in functions *f3* and *f4* where call *a* occurs in an earlier expression to the expression that contains call *b*. This gives it a support of four. In a similar manner the pattern $\{c\}\rightarrow\{b\}$ occurs in the function *f2*, *f3* and *f4* giving it a support of three. These two patterns are totally ordered, whereas, the third pattern, $\{a, c\}\rightarrow\{b\}$, is only partially ordered. This is due to the calls *a* and *c* occurring in the same expression that is an argument to the call *b*. This only occurs in functions *f2* and *f3*, so the support is two.

```
void f1(){                void f2(){
 d();                      …
 b(x+a());                 b(c()+a());
 …                         k();
}                         }

void f3(){                void f4(){
 e();                       x=a();
 y=c()+a();                 y=c();
 b(y);                      b(x+y);
}                         }
```

Figure 1. An example of four function definitions with calls to functions *a*, *b*, and *c* for demonstrating patterns, variants, and violations

From these patterns two rules can be formed. The first rule is $\{a\}\rightarrow\{b\}\Rightarrow\{a, c\}\rightarrow\{b\}$, i.e., when there is a call *a* followed by a call *b*, then a call *c* occurs in the same expression as the call *a*. Of the four functions that contain the pattern $\{a\}\rightarrow\{b\}$ only two contain the pattern $\{a, c\}\rightarrow\{b\}$ producing an association rule with a confidence of 0.5. The second rule is $\{c\}\rightarrow\{b\}\Rightarrow\{a, c\}\rightarrow\{b\}$ with a confidence of 0.67 since the rule applies to two out of the three functions that contain the pattern $\{c\}\rightarrow\{b\}$. The ordered patterns $\{a\}\rightarrow\{b\}$ and $\{c\}\rightarrow\{b\}$ are the only two order-preserving subsets of the ordered pattern $\{a, c\}\rightarrow\{b\}$ that form sequence rules. However only the second rule satisfies the required minimum confidence. As a result the ordered patterns $\{c\}\rightarrow\{b\}$ is reported as a variant in functions *f4*.

We have applied both sequential-pattern and itemset mining on the Linux kernel v2.6.14. As examples we will use some of the patterns, variants, and violations uncovered from this system. The unordered pattern *{spin_lock_irqsave, spin_unlock_irqrestore}* occurs in over two thousand functions. This pattern suggests that the calls *spin_lock_irqsave* and *spin_unlock_irqrestore* are typically used together and is considered to be a candidate usage pattern. However, there are seventeen functions in which the call *spin_lock_irqsave* occurs without the call *spin_unlock_irqrestore.* Therefore, the call *spin_lock_irqsave* is reported as a candidate variant. This variant forms seventeen violations. For example the function *esp_open* in the file *drivers/char/esp.c* contains the violation (*spin_lock_irqsave*, *drivers/char/esp.c#esp_open*) of the pattern *{spin_lock_irqsave, spin_unlock_irqrestore}*. This violation indicates that the call *spin_unlock_irqrestore* is missing in the function *esp_open.*

The sequential-pattern mining produces the ordered pattern *{spin_lock_irqsave}→{spin_unlock_irqrestore}* which occurs the same number of times as the above unordered pattern. This pattern suggests that not only the calls *spin_lock_irqsave* and *spin_unlock_irqrestore* occur together but they also have a specific order. The sequential violation (*spin_lock_irqsave*, *drivers/char/esp.c#esp_open*) indicates either the call *spin_unlock_irqrestore* is missing (as in the itemset violation) or occurs before the call *{spin_lock_irqsave}* in the function *esp_open.*

2.4. Itemset versus Sequence Mining

We now contrast the two approaches with regards to their underlying methodology and characteristics of the uncovered patterns. Both itemset and sequential-pattern mining approaches are driven by their support mechanism for establishing a set of calls in a function as a candidate pattern. In itemset-pattern mining a binary check for presence or absence of all the constituent calls in a function is sufficient to count that function towards its support. The order of calls is completely ignored in mining. In sequential-pattern mining an additional constraint of ordering is required. A function only counts towards the support of a pattern if all the constituent calls are found in the exact same order as in the pattern. Therefore, itemset mining operates on a more relaxed constraint of appearance only than sequential-pattern mining that needs both appearance and order. Itemset mining is a generalized approach, whereas, sequence mining is a specialized approach.

The generalized itemset mining approach affects the coverage with regards to types of patterns and variants. Let us look at each separately.

Variant multiplicity: The binary-check approach for counting support ignores the number of times the constituent call is present in a function. If a variant occurs due to multiple call occurrences, they are left uncovered. For example, calls appearing as *lock*, *unlock*, and *lock* with the second call to *lock* missing a matching call to *unlock.*

Out-of-Order Variants: Since the order of calls in a function is completely ignored, variants that occur due to incorrect ordering of calls (e.g., potential bugs or non-standard usage of a call composition) will not be uncovered by itemset mining. Additionally, out-of-order variants may result in false reporting of standard (possibly larger) unordered patterns as order is ignored. This is due to the overgeneralization of multiple variants into a single pattern. The variants *{a}→{b}* and *{b}→{a}* with support of 5 and 10 respectively would be reported as a subset *{ab}* with a support of 15.

Context Information of variants: Assume that a variant is a true bug due to a missing call(s). In this case, the only information available to the external user (e.g., a developer or a tool) from a variant pattern is the calls that are missing but not the order in which they should be inserted to fix the bug, or in the case of multiple calls to the same function which particular instances of the calls are part of the pattern.

3. Comparing the Two Techniques

Ideally one would like to compare the itemset-mining and sequence-mining approaches directly in terms of their effectiveness in solving a particular task. Bug location is one such task that has been previously performed with itemset mining [7]. Unfortunately, such a comparison is not feasible with regards to a large system (e.g., Linux with over 6,000 KLOC) in a reasonable time period. This is primarily due to candidate patterns and variants reported in the order of hundreds or thousands from large software systems. Manual examination of all the candidate patterns is not practical. The lack of documentation of standard usages negates another source to establish a comparison baseline. While the examination of version history is a possible source of validation, variants may go unnoticed for a number of versions due to their latent nature. They may only begin to be noticed after they have affected the maintainability of the system or a severe bug is discovered. Without a clear comparison there is very little benefit in employing traditional validation metrics such as precision and recall.

One possible approach is a comparison based on the number of patterns, functions with variants, and violations. Frequent-pattern mining produces a large number of patterns from a large-scale software system. Typically, many of the violations derived from these patterns are false positives (e.g., a violation is reported as a potential bug but is not a bug). A technique that produces much fewer false violations and variants is more desirable. One way to achieve this is to prefer a technique that produces much fewer patterns and thus possibly fewer false variants and violations. However, this approach imposes a risk of discarding valid patterns, i.e., false negatives. A technique that reduces the number of false-positive violations as well reduces the number of false-negative patterns and violations are more desirable with respect to the overall accuracy. These measures also have a direct impact on the number of lines of codes that a developer has to examine and/or an additional analysis tool has to process. This compounded with the false-positive issue, makes these measures reasonable indicators of the effectiveness of an approach.

Table 1. Linux call sequence statistics.

System	KLOC	Number of functions	Number of calls	Avg. calls/function
Linux kernel (v2.6.14)	6,304	112,671	806,297	7.2

Ordered patterns provide more context information to the application or user at the expense of a higher cost. In the case of an itemset mining, the search space of all possible unordered patterns with *n* different call usages in a system is 2^n. However, sequential-pattern mining has to potentially consider a combinatorial explosion in the search space of all possible patterns of the order of $\Theta(2^{nm})$ with *m* partially ordered components each consisting of an average of *n* calls. Therefore, a sequential-pattern mining technique could be much more computationally expensive. In practice we found that our implementation of itemset mining took about 62 minutes for a minimum support of 20 (the most time consuming) on the Linux kernel and sequence mining took around 241 minutes. For the higher values both took much less time. Therefore, for lower support values, the cost of sequential pattern mining is approximately four times the cost of itemset mining.

The computation time is really less of an issue as mining will be a relatively infrequent activity compared to inspection of the candidate violations. Therefore, it is the number of variants, functions with variants, and variations that are of serious concern.

3.1. Evaluation on Linux Kernel

In order to facilitate the comparison, we applied both mining techniques on the Linux kernel v2.6.14. First the ordered patterns from all functions were extracted. The statistics of the considered code base along with the numbers of functions and calls are shown in Table 1.

We developed the tool *callextractor* for extracting call sequences based on the *srcML* format (www.sdml.info/projects/srcml) and the tool *sqminer* for mining frequent patterns. We used *sqminer* for mining the ordered patterns, variants, and violations directly from the ordered patterns extracted by *callextractor*. In addition the tool *sqminer* was configured for itemset mining to mine the unordered patterns, variants, and violations from the unordered patterns formed from the ordered patterns extracted by *callextractor*. Since the results of frequent-pattern mining are sensitive to the externally supplied minimum-support value, six runs of *sqminer* were performed with different minimum-support values and a minimum confidence of 0.9 on a Pentium 4, 3.0GHZ machine with 1GB RAM. The minimum support was doubled in each successive run starting with 20. The number of patterns, variants, functions with variants, and violations are given in Table 2. The following observations can be made:

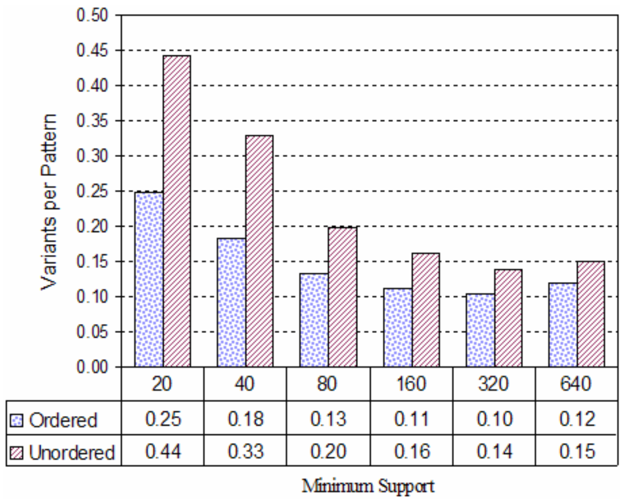
- Sequence mining found more patterns than itemset mining for all minimum-support values.
- Sequence mining found fewer variants than itemset mining for most minimum-support values.
- Sequence mining found less violations than itemset mining for minimum-support values of 20 and 40, whereas, for minimum-support values 80 160, 320, and 640 the opposite occurs.

Table 2. A comparison of sequential-pattern mining (ordered) and itemset-pattern mining (unordered) approaches for Linux kernel v2.6.14. Violations are the total number of (variant, function) pairs.

	Minimum support	Number of patterns	Number of variants	Number of functions with variants	Violations
Sequence Mining	20	35832	8907	4652	30284
	40	9657	1760	3023	10156
	80	2883	381	2024	4700
	160	996	111	1459	2669
	320	356	37	1018	1558
	640	143	17	754	1089
Itemset Mining	20	25813	11404	3736	57908
	40	7514	2464	2508	15847
	80	2132	421	1697	4093
	160	715	116	1188	2159
	320	254	35	861	1204
	640	106	16	666	840

The number of patterns and variants decrease with increase in minimum support. The patterns mined with low support values are more likely to be reflective of functions with more specific functionality, whereas, patterns with much higher support may reflect ubiquitous functions that are used throughout the system. The number of variants is lower in the case of ordered patterns in the range of hundreds to thousands for the minimum-support values of 20 and 40. Furthermore, the number of violations is lower for ordered patterns in the range of thousands and as much as 1.5 times. Overall, there are more ordered patterns and fewer variants for sequential-pattern mining than for itemset mining in majority of the minimum support runs. For lower support values of 20 and 40 the number of violations of ordered patterns is less than that of unordered patterns. This suggests that sequential-pattern mining could reduce false positives of variants and violations without compromising false negatives of ordered patterns.

The ratio of number of variants to the number of patterns gives a general idea as to how many of the patterns are overall violated. Figure 2 shows that this ratio is lower in all cases of ordered patterns. For a minimum-support value of 20, itemset mining reports 44% of the patterns have variants (i.e., 56% of the patterns are followed) and sequential-pattern mining reports 25% of the patterns have variants (i.e., 75% of the patterns are followed). If variants are used as bug indicators, itemset mining would report more candidate bugs than sequential-pattern mining. As such sequential-pattern mining generally uncovers more potential patterns and reduces the number of variants.

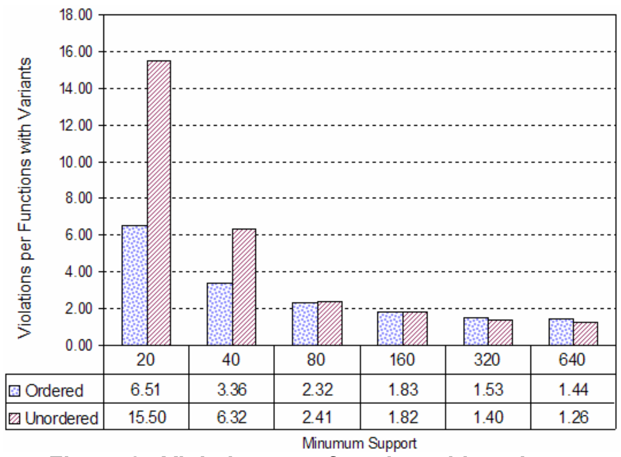


	20	40	80	160	320	640
▩ Ordered	0.25	0.18	0.13	0.11	0.10	0.12
▨ Unordered	0.44	0.33	0.20	0.16	0.14	0.15

Figure 2. Variants per pattern showing that sequential-pattern mining outperforms itemset mining as the ratio of the number of variants per pattern is always low for ordered patterns.

Another useful measure is the ratio of the number of violations to the number of functions containing variants. The results of this measure are shown in Figure 3. Once again, sequential-pattern mining outperforms itemset mining for lower support values and performs equally well for higher values. For a minimum support of 20, sequential-pattern mining would report on average approximately seven violations per function with variants, whereas, itemset-pattern mining would report on average fifteen violations per function with variants, i.e., a ratio more than two times higher. The above results indicate that sequential-pattern mining generally produces a substantially lower number of variants and violations compared to itemset mining.

The size of patterns in terms of the number of calls gives us, at least, a cursory idea about the complexity of the typical call usages and its impact on variants (e.g., a larger pattern may mean more possibility of its violation). Also, the ordering of calls in a pattern may become more desirable with increase in size. For a minimum support of 20, the largest pattern is composed of 16 calls in both itemset and sequential-pattern mining. Overall, itemset mining produces a higher number of larger patterns and a lesser number of smaller patterns than sequential-pattern mining. The singleton and binary patterns make up approximately 27% and 30% of the total patterns in sequential-pattern and itemset mining respectively.



	20	40	80	160	320	640
▩ Ordered	6.51	3.36	2.32	1.83	1.53	1.44
▨ Unordered	15.50	6.32	2.41	1.82	1.40	1.26

Figure 3. Violations per function with variants showing that sequential-pattern mining outperforms itemset mining for lower support values and performs equally for higher support values.

3.2. Validation

Our interest is in what these variants mean and what tasks they could support. Our first comparison between the two techniques is in regard to a potential bug in an older version of the Linux kernel (v2.6.11) reported by Li et al [7]. The bug in question is the missing call *scsi_scan_host* in the function *sbp2_alloc_device* in the file *sbp2.c* that violates the unordered pattern *{scsi_host_alloc, scsi_add_host, scsi_scan_host}*. This violation is due to the (association) rule *{scsi_host_alloc, scsi_add_host}* $\Rightarrow$ *{scsi_scan_host}* which has a confidence of 0.93. The smaller unordered pattern *{scsi_host_alloc, scsi_add_host}* occurs in 29 functions while the larger unordered *{scsi_host_alloc, scsi_add_host, scsi_scan_host}* occurs in 27 functions.

A direct comparison of previous itemset-mining results in [7] with that of sequential-pattern mining is not feasible due to lack of insufficient information in reproducing the exact evaluation setup. Therefore, the results of itemset and sequential-pattern mining are compared with regards to the above violation in the v2.6.14. The following results were obtained,

Itemset mining: The association rule *{scsi_host_alloc, scsi_add_host}* $\Rightarrow$ *{scsi_scan_host}* was reported with the confidence of 0.83, the unordered pattern *{scsi_host_alloc, scsi_add_host}* occurs in 42 functions, and the unordered pattern *{scsi_host_alloc, scsi_add_host, scsi_scan_host}* occurs in 35 functions.

Sequential-pattern mining: The sequence rule *{scsi_host_alloc}→{scsi_add_host} ⇒ {scsi_host_alloc} → {scsi_add_host}→{scsi_scan_host}* was reported with the confidence of 0.83, the ordered patterns *{scsi_host_alloc} → {scsi_add_host}* and *{scsi_host_alloc}→{scsi_add_host} → {scsi_scan_host}* occur in 42 and 35 functions respectively.

Both itemset and sequential-pattern mining are equally likely to report this violation as a bug as they have the same confidence for the rules. However, ordered pattern gives the precise location of where the missing call should be. The corresponding unordered pattern only tells that this call is missing and not the precise location at where it should have been or should be added. Note that implication of association rules in case of itemset mining does not means the order in which calls should occur. It simply tells that whenever the calls *scsi_host_alloc* and *scsi_add_host* occur, the call *scsi_scan_host* should also occur.

We manually inspected the function *sbp2_alloc_device* in version 2.6.14 and were not able to confirm the above violation as a bug. So we examined all versions up to version 2.6.16 and found that the call *scsi_scan_host* was still absent in the function *sbp2_alloc_device*. This indicated to us that the above violation is potentially a false positive or it is a bug that has not been fixed yet. In any case, it is safe to surmise that the accuracy of both itemset and sequential-pattern mining is the same.

In order to further examine the results of sequential-pattern mining, we analyzed the functions that were reported to have variants with a minimum support of 20 found by sequence mining but not by itemset mining. Of 1,895 functions with violations, 389 (approximately 20%) were order violations, i.e., all of the calls are present but their composition is not in the right order. Of these order violations, 65 ordered patterns were changed in the later version of the Linux kernel v2.6.16.20. This validates that there are ordering violations that only sequential-pattern mining uncovers.

One example of ordered pattern with an order violation is in the function *adi_connect* in the file *drivers/input/joystick/adi.c* which contains the ordered pattern *{gameport_set_drvdata}→ {kfree}→ {gameport_close}→{kfree}*. This pattern is violated by multiple variants including the following three rules,

- *{gameport_close}→{kfree}⇒{gameport_close} →{gameport_set_drvdata}→{kfree}*,
- *{gameport_set_drvdata}⇒{gameport_close}→ {gameport_set_drvdata}→{kfree}*, and
- *{gameport_set_drvdata}→{kfree}⇒{gameport_close}→{gameport_set_drvdata}→{kfree}*.

In a later version of the Linux kernel (v 2.6.16.20) an additional call *gameport_set_drvdata* was added to this same function creating a new ordered pattern *{gameport_set_drvdata}→{kfree}→{gameport_close}→ {gameport_set_drvdata}→ {kfree}*. As a result, all of the above rules were no longer under violation. This indicates that this violation was a potential bug or a non-standard usage. This example demonstrates that sequential-pattern mining is able to find violations that are not uncovered by itemset mining.

4. Related Work

First, we discuss the work related to the problem of finding usage patterns and then the use of frequent-pattern mining methodologies in software engineering for some other purposes. This list is by no means exhaustive but represents a number of different investigations.

Michail [10] presented an approach based on itemset and association-rule mining to uncover entities such as components, classes, and functions that occur frequently together in library usages. Similar to the work presented here, Li et al [7] addresses the question of extracting rules and violations of typical usages of function calls in a system. Their approach is based on itemset mining. They show the application of their approach in bug location. Their call extraction uses the *gcc* front end, whereas, our call-extracting mechanism is based on the language standards and decoupled from a specific compiler implementation.

Livshits and Zimmermann [8] present an approach based on itemset mining for discovering call-usage patterns from source-code versions. They classified the mined patterns into valid patterns, likely error patterns, and unlikely patterns with additional dynamic analysis. Williams et al [13] analyzed usages of function-return values for detecting software bugs via static analysis of a single version and evolutionary changes. A number of researchers used a combination of static and dynamic analyses, and finite state automaton to infer usage patterns and program properties. [2, 11, 12, 15].

Itemset mining and sequential-pattern mining techniques have been applied to other some problems in software engineering. Zimmerman et al [17] used *CVS* logs for detecting evolutionary coupling between source-code entities. Yang et al [16] used a similar technique for identifying files that frequently change together. Burch et al [3] presented a tool that supports visualization of association rules and sequence rules. El-Ramly et al [4] used sequential-pattern mining to detect patterns of user activities from system-user interaction data. Kagdi et al [6] used sequence mining to extract a sequence of co-changed files from source-code repositories. Xie et al [14] used sequence mining to filter the results of a source-code search tool to report API-usage patterns in which a source-code entity is used. However, a sequence-mining approach has not been used before for function-call usage patterns discovery.

5. Conclusions and Future Work

We compared itemset and sequential-pattern mining with regards to the number of patterns, variants, and violations when applied to a large system. Our results show that itemset mining produces unordered patterns that are larger in size and higher in support, with more violations when compared with sequential-pattern mining. Itemset mining's over-generalized behavior results in more false positive candidates than sequential-pattern mining. Sequential-pattern mining produces smaller patterns with the additional benefit of ordering information. We identified candidate violations that were not found via itemset mining. One such case was demonstrated as a part of our validation. The computational cost of sequential-mining is higher than itemset mining. However, this cost is compensated for the time saved in examining fewer false positives and covering more valid cases.

Comparison metrics as the "gold standard" for validating the violations remains an important and difficult issue. One promising source is the version history as was used in our validation. However, this may not be sufficient due to the latent nature of many of the patterns and their violations that may go unnoticed for a number of versions.

We are currently analyzing call patterns taking into account conditional and iterative constructs. We plan to compare these two techniques on a number of open-source systems, in conjunction with (and without) static and dynamic analysis techniques. We are also extending our call-extraction tool to include other languages such as C++ and Java. A major extension with regards to call extraction in object-oriented languages is the need for analysis of calls via inheritance and polymorphism.

6. References

[1] Agrawal, R. and Srikant, R., "Mining Sequential Patterns", in Proceedings of 11th International Conference on Data Engineering, Taipei, Taiwan, March 1995.

[2] Ammons, G., Bodik, R., and Larus, J. R., "Mining specifications", in Proceedings of 29th Symposium on Principles of Programming Languages (POPL'02), Portland, OR, January 16-18 2002, pp. 4-16.

[3] Burch, M., Diehl, S., and Weißgerber, P., "Visual Data Mining in Software Archives", in Proceedings of ACM Symposium on Software Visualization (SoftVis'05), St. Louis, Missouri, May, 14-15 2005, pp. 37-46

[4] El-Ramly, M. and Stroulia, E., "Mining Software Usage Data", in Proceedings of 1st International Workshop on Mining Software Repositories (MSR'04), 2004, pp. 64-8.

[5] Goethals, B., "Frequent Set Mining", in *The Data Mining and Knowledge Discovery Handbook*, Springer, 2005, pp. 377-397.

[6] Kagdi, H., Yusuf, S., and Maletic, J. I., "Mining Sequences of Changed-files from Version Histories", in Proceedings of 3rd International Workshop on Mining Software Repositories (MSR'06) Shanghai, China, May 22-23, 2006, pp. 47-53

[7] Li, Z. and Zhou, Y., "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code", in Proceedings of 13th International Symposium on Foundations of Software Engineering (ESEC/FSE'05), Lisbon, September 2005, pp. 306-315

[8] Livshits, B. and Zimmermann, T., "DyanMine: Finding Common Error Patterns by Mining Software Revision Histories", in Proceedings of 13th International Symposium on Foundations of Software Engineering (ESEC/FSE'05), Lisbon, Portugal, September 2005, pp. 296-305

[9] Masseglia, F., Teisseire, M., and Poncelet, P., "Sequential Pattern Mining: A Survey on Issues and Approaches ", in *Encyclopedia of Data Warehousing and Mining* Information Science Publishing, 2005.

[10] Michail, A., "Data Mining Library Reuse Patterns Using Generalized Association Rules", in Proceedings of 22nd International Conference on Software Engineering (ICSE'00), Limerick, Ireland, June 4-11 2000, pp. 167-176.

[11] Nimmer, J. W. and Ernst, M. D., "Invariant Inference for Static Checking: An Empirical Evaluation", in Proceedings of 10th International Symposium on the Foundations of Software Engineering (FSE '02), Charleston, Nov 20-22 2002, pp. 11-20.

[12] Whaley, J., Martin, M. C., and Lam, M. S., "Automatic Extraction of Object-Oriented Component Interfaces", in Proceedings of International Symposium on Software Testing and Analysis (ISSTA'02), Rome, Italy, 2002, pp. 218-228.

[13] Williams, C. C. and Hollingsworth, J. K., "Recovering System Specific Rules from Software Repositories", in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05) St. Louis, 2005 pp. 7-11

[14] Xie, T. and Pei, J., "MAPO: Mining API Usages from Open Source Repositories", in Proceedings of 3rd International Workshop on Mining Software Repositories (MSR'06) Shanghai, China, May 22-23, 2006 2006, pp. 54-57.

[15] Yang, J., Evans, D., Bhardwaj, D., Bhat, T., and Das, M., "Perracotta: Mining Temporal API Rules from Imperfect Trace", in Proceedings of 28th International Conference on Software Engineering (ICSE'06), Shanghai, China, May 20-28 2006, pp. 282-291.

[16] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C., "Predicting Source Code Changes by Mining Change History", *IEEE Transactions on Software Engineering*, vol. 30, no. 9, September 2004, pp. 574 - 586

[17] Zimmermann, T., Zeller, A., Weißgerber, P., and Diehl, S., "Mining Version Histories to Guide Software Changes", *IEEE Trans on Software Eng.*, vol. 31, no. 6, 2005, pp. 429-445.