2014 IEEE 38th Annual International Computers, Software and Applications Conference

BuildPredictor: More Accurate Missed Dependency Prediction in Build Configuration Files

Bo Zhou*, Xin Xia*‡, David Lo†, and Xinyu Wang*§
*College of Computer Science and Technology, Zhejiang University
†School of Information Systems, Singapore Management University
xxkidd@zju.edu.cn, davidlo@smu.edu.sg, {xinyuwang, bzhou}@zju.edu.cn

***Abstract*—Software build system (e.g., make) plays an important role in compiling human-readable source code into an executable program. One feature of build system such as make-based system is that it would use a build configuration file (e.g., Makefile) to record the dependencies among different target and source code files. However, sometimes important dependencies would be missed in a build configuration file, which would cause additional debugging effort to fix it. In this paper, we propose a novel algorithm named *BuildPredictor* to mine the missed dependncies. We first analyze dependencies in a build configuration file (e.g., Makefile), and establish a dependency graph which captures various dependencies in the build configuration file. Next, considering that a build configuration file is constructed based on the source code dependency relationship, we establish a code dependency graph (code graph). *BuildPredictor* is a composite model, which combines both dependency graph and code graph, to achieve a high prediction performance. We collected 7 build configuration files from various open source projects, which are Zlib, putty, vim, Apache Portable Runtime (APR), memcached, nginx, and Tengine, to evaluate the effectiveness of our algorithm. The experiment results show that compared with the state-of-the-art link prediction algorithms used by Xia et al., our *BuildPredictor* achieves the best performance in predicting the missed dependencies.**

***Keywords*—Build System, Link Prediction, Build Graph, Code Graph, Makefile**

I. Introduction

In a modern software, build system is an indispensable component. Build systems compile source code, libraries and other data into executable programs by orchestrating the execution of different compilers and other tools [1]. There are various build systems, such as **make**, **ant**, **scon**, **cmake**, and **maven**, and they work on different programming languages and platforms. Due to the complexity of modern software systems, the maintenance of build systems is a difficult job. A prior research study has shown that build maintenance could add 12%-36% additional costs to software development [2].

In the build process, a build system (e.g., **make**) first reads a build configuration file (e.g., Makefile), and then executes the rules in the configuration file to build a system. These rules in the build configuration file represent the dependencies among different targets and source code files. However, for a large-scale software project, such as Linux, the dependencies among the targets and source code files are complex. Thus, it is easy to miss some dependencies, which is hard to detect.

‡The work was done while the author was visiting Singapore Management University.
§Corresponding author.

To address the dependency mining problem, Xia et al. first reverse engineer a build configuration files (e.g., Makefile) into a dependency graph where nodes in the graph correspond to entities in the Makefile and edges in the graph correspond to relationships among these entities, and then map the problem into a link prediction problem which would predict the missed edges (links) in the dependency graph [3].

In this paper, we further investigate the dependency mining problem. Similar with Xia et al., we focus on *make* build tool, since it is one of the most widely used build tools. Our goal is to improve the effectiveness of the nine link prediction algorithms used by Xia et al. [3] which is still low. The algorithms used by Xia et al. only analyze build configuration file and ignore source code to infer missed dependencies. Furthermore, the nine link prediction algorithms are originally proposed to solve social network and biological problems [4], and they ignore some basic rules that are obeyed by dependencies in build configuration files. To overcome the limitations of Xia et al.'s approach for dependency mining, we propose *BuildPredictor* to achieve a better performance.

BuildPredictor combines two graphs – dependency graph which is constructed from the dependencies in a build configuration file, and code graph which is constructed from the dependencies in source code files – to predict the missed dependencies. It then computes a number of similarity scores between pairs of nodes in the dependency and code graphs. These similarity scores are used to rank candidate pairs of nodes in the dependency graph that are not connected together with an edge. *BuildPredictor* leverages a number of rules that are observed by build configuration files to prune candidate pairs (a, b) where it is impossible or highly unlikely that an edge exists between a and b. *BuildPredictor* would then recommend the remaining top pairs for developers to check to identify missed dependencies.

To evaluate the benefits of *BuildPredictor*, we collect 7 build configuration files from various open source projects, such as Zlib, putty, vim, Apache Portable Runtime (APR), memcached, nginx, and Tengine. *BuildPredictor* achieves an average top-n precision and AUC score of 80.38% and 0.8512 across the 7 projects. Top-n precision measures the accuracy of the top few recommendations made by a dependency mining approach, while AUC measures the overall accuracy of the entire recommendations. Since developers are likely to investigate only top few recommendations and would likely not use the tool if the top few recommendations are bad (c.f., [5]), achieving high top-n precision is more important than high AUC. The experiment results show that our *BuildPredictor*

```
calculator :add.o subtract.o mult.o divide.o lib.a
      gcc add.o subtract.o mult.o divide.o -L . lib.a -o
calculator

add.o: add.c num.h add.h
      gcc -c add.c

subtract.o: subtract.c num.h add.h
      gcc -c subtract.c

mult.o: mult.c num.h mult.h
      gcc -c mult.c

divide.o: divide.c num.h mult.h
      gcc -c divide.c

clean:
      rm -rf *.o
```

Fig. 1. An Example Makefile which Specifies the Build Process for `calculator`.

outperforms the nine link prediction algorithms used by Xia et al. [3] substantially, especially in terms of top-n precision. The highest performing link prediction algorithm only achieves a top-n precision and AUC of 1.73% and 0.83.

II. Preliminaries and Problem Definition

make build tool uses a specific configuration file called Makefile, and the goal of a Makefile is to tell **make** tool how to build a system. A Makefile constitutes of a number of rules. The rules have a similar structure [1]:

$$\begin{aligned} target... \quad &: prerequisites... \\ &commands \end{aligned} \tag{1}$$

The $target$ can be an object file, an executable file, or even a label. $prerequisites$ specify source code files, object files or executable file that are needed, or other targets that need to be processed to build $target$. Thus $prerequisities$ store dependencies between a target to other targets or source code files. $commands$ tells the build system how to generate $target$ from $prerequisites$.

Figure 1 presents an example of Makefile which specifies the build process of a calculator. There are 6 $targets$ in the Makefile: *add.o, subtract.o, mult.o* and *divide.o* are object files; *calculator* is an executable file; *clean* is a label, since its $prerequisites$ is empty. *calculator* target depends on several $prerequisites$: *add.o*, *substract.o*, *mult.o*, *divide.o*, and a static library *lib.a*. The object files (e.g., *add.o, subtract.o, mult.o* and *divide.o*) depends on their own source code files and a common header file *num.h*.

By analyzing Figure 1, we could create a dependency graph. Figure 2 shows the dependency graph derived from the example Makefile shown in Figure 1. The $target$ and $prerequisites$ in the Makefile become nodes, and the dependencies become edges in the dependency graph. Note that the dependency graph is a directed acyclic graph (DAG), and it is also a rooted graph with calculator as the root note. The formal definition of dependency graph is shown in Definition 1. Based on this graph, we formally define dependency mining problem in Definition 2.

Definition 1: (**Dependency Graph.**) We denote a dependency graph as $G(V, E)$, where each node $v \in V$ corresponds to a $target$ or a member of a $prerequisites$ list in a corresponding build configuration file (e.g., Makefile), and each link $e \in E$ denotes the dependency relationships between a $target$ and its $prerequisites$. Since the $prerequisites$ of a $target$ can have multiple members, a $target$ node can be linked to more than one node in a dependency graph. Notation-wise, we use $e(v', v)$ to denote a link from node v' to node v.

Definition 2: (**Dependency Mining.**) Given a dependency graph $G(V, E)$, let us denote the set containing all the $\frac{|V|\times(|V|-1)}{2}$ edges in the graph as U. Then, the dependency mining problem is to recover the edges in the $U - E$ edges that correspond to missed dependencies.

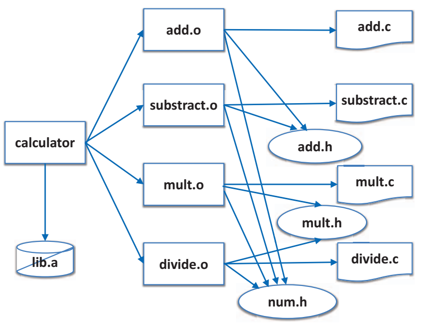

Fig. 2. Dependency Graph for the Example of Makefile in Figure 1

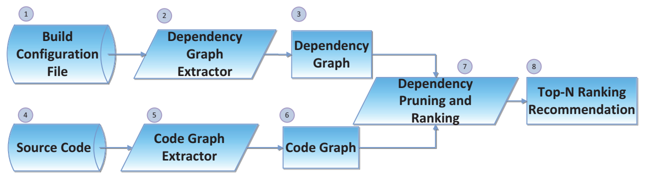

Fig. 3. Overall Framework of *BuildPredictor*.

III. Our Proposed Approach

In this section, we present our proposed approach named *BuildPredictor*. We first describe our overall framework. Then we propose the way to extract dependency graph and code graph, respectively. Next, we describe the dependency pruning and ranking method.

Overall Framework. We present the whole framework for *BuildPredictor* in Figure 3. Our approach takes as input a build configuration file (1) and source code files (4). The configuration file is input to the dependency graph extractor component (2) which analyzes the build configuration file and outputs a dependency graph (3). The source code files are analyzed by the code graph extractor (5) which eventually produces a code graph (4). The generated dependency and code graphs are then input to the dependency pruning and ranking component (7) which eventually produces a list of top-N candidate missed dependencies sorted based on their likelihood to be true.

Dependency Graph Extraction. To extract a dependency graph from a build configuration file we make use of MAKAO [6]. MAKAO takes in a build configuration file and outputs a dependency graph whose definition is given in Definition 1.

We divide the nodes in dependency graph into two types, i.e., source nodes and target nodes. Source nodes refer to the files which exist before we build a system, which include different types of files, such as source code files, configuration files, etc. Target nodes refer to the targets which appear after we run **make** command to build a system, such as the object files, and executables. Definition 3 formally define source and target nodes. It makes use of the concept of out-neighbors and out-degree which are defined in Definition 4.

Definition 3: (**Source Node and Target Node.**) Given a dependency graph $G(V, E)$, a node v is a source node if it has zero out-neighbor, i.e., $k_{out}(v) = 0$; a node v is a target node if it has at least one out-neighbors, i.e., $k_{out}(v) \geq 1$.

Definition 4: (**Out-neighbors and Out-degree of a Node.**) Given a dependency graph $G(V, E)$, and a node $v \in V$, the out neighbors of v is a set of nodes v' which $e(v, v')$ exist. Let $\Gamma_{out}(v)$ denotes the out neighbors in G. And the out-degree of v is the total number of out-neighbors of v. Let $k_{out}(v)$ denotes the degree of node v in G.

$$k_{out}(v) = |\Gamma_{out}(v)| \tag{2}$$

Code Graph Extraction. A code graph is a homogeneous network, i.e., each node in the code graph has the same type, and each node corresponds to a source code file. In C/C++ language, there are various types of source code files, such as .c, .h, .cpp and .hpp. In this paper, we do not differentiate them. To extract a code graph, we analyze the source code files and extract the "include" statement from C/C++ source code files. We search the whole project space to build the mapping of source code files to the files that they depend on, and we ignore files which are in the standard C/C++ library.

Algorithm 1 Dependency Pruning and Ranking Method.

```
1: BuildPredictor($depGraph$, $codeGraph$, $buildNodes$, $codeNodes$)
2: Input:
3: $depGraph$: dependency braph
4: $codeGraph$: code graph
5: Output: top-n missed dependencies
6: Method:
7: Let $targetNodes$ = Target nodes in $depGraph$ (see Definition 3)
8: Let $sourceNodes$ = Source nodes in $depGraph$ (see Definition 3)
9: Let $codeNodes$ = Nodes in the $codeGraph$
10: Map nodes in $codeNodes$ to $sourceNodes$;
11: Infer hierarchical structure of $depGraph$, following Algorithm 2;
12: Let $candidateMissed$ = {p|p = a pair of unlinked nodes in $depGraph$};
13: Prune node pairs in $candidateMissed$ which violate Rule 1, 2 and 3;
14: Compute code similarity scores according to Definition 6;
15: Compute target to target node similarity scores using to Algorithm 4;
16: Compute target to source node similarity scores using Algorithm 3;
17: Rank target-to-target links, and target-to-source links in $candidateMissed$ based
    on the similarity scores;
18: Return top-n missed dependencies in $candidateMissed$;
```

Definition 5: (**Code Graph.**) We denote a code graph as $G(V, E)$, where each node $v \in V$ denotes a source code file, and each link $e \in E$ denotes the dependencies between the node pairs. For each source code file v, we extract its "include" statements to find the source code files it depends on. We create a link from source code file v to each of the source code files that it depends on. We denote the link from node v to node v' as $e(v, v')$.

Dependency Pruning and Ranking. After the dependency and code graphs are constructed, we analyze dependencies that are missed in the dependency graph. We then prune improbable dependencies based on some rules that build system obeys. We then rank the remaining missed dependencies based on their likelihood to be true. The goal is to rank real missed dependencies (i.e., these dependencies are erroneously missed) high in the ranked list.

Our algorithm to prune and rank dependencies is shown in Algorithm 1. The algorithm begins by extracting the set of source and target nodes from the dependency graph (Lines 7-8). It then also extracts the set of nodes from the code graph (Line 9). The algorithm then proceeds to map nodes in source nodes with nodes in the code graph (Line 10). It then infers the hierarchical structure of the dependency graph (Line 11). This process assigns to each node its unique level, which corresponds to its distance to the root of the dependency graph. After the hierarchical structure is inferred, we get the set of missed dependency candidates $candidateMissed$ which correspond to each pair of nodes in the dependency graph that are not linked to each other (Line 12). We then prune some of the dependencies in $candidateMissed$ that are improbable based on some basic rules that build configuration files obey (Line 13). Next, we compute the similarity score between two nodes in the code graph (Line 14). These similarity scores are then used to compute the similarity between one target node to another target node, and one target node to a source node in $candidateMissed$ (Lines 15-16). These scores are then used to rank dependencies in $candidateMissed$ (Line 17). The top-n dependencies with the highest scores are then output.

In the next few subsections, we elaborate the process to infer hierarchical structure, prune dependencies that violate some rules, compute code similarity, compute target-to-source node similarity, and compute target-to-target node similarity.

1) Infer Hierarchical Structure: A dependency graph has a hierarchical structure. There is a root node v, which is at level 0. For the root node v, there is no other node v' where $e(v', v)$ exists. Nodes that are directly connected to the root node v is at level 1. The highest levels in a dependency graph are source nodes, i.e., their out-neighbor is 0. The procedure to infer the levels of each node in a dependency graph is shown in Algorithm 2. We first identify the root node in the dependency graph (Line 7). Next, the out-neighbors of the root node are the level 1 nodes (Line 8). We iterate the process until all the nodes in dependency graph have their assigned levels (Lines 9-11). As an example, consider the graph shown in Figure 2, $calculator$ is the root node, and there are 3 levels (i.e., 0, 1, and 2) in the dependency graph.

2) Dependency Pruning: In the dependency pruning step, we remove links in $candidateMissed$ that violate one of the following rules:

Rule 1: (**Redundant Link Rule.**) The link between node v and v' is not a missed dependency if there exists a path in the dependency graph that connects v and v'.

Rule 2: (**Source Nodes Rule.**) No link would exist among source nodes, and no link would exist from a source nodes to a target node, i.e., only two types of links exist in a dependency graph: links from target nodes to target nodes, and links from targets nodes to source nodes.

Algorithm 2 Hierarchical Structure Inference

```
1: InferHierarchicalStructure($graph$, $nodes$)
2: Input:
3: $graph$: dependency graph
4: $nodes$: nodes in $graph$
5: Output: The level of each node in $nodes$
6: Method:
7: Find the root node $v \in nodes$ where there exists no other nodes $v'$ where $e(v', v)$ exists;
8: Find the level-1 node sets $V_{level1} = \{v|v \in \Gamma_{out}(v)\}$;
9: Find the level-2 node sets $V_{level2} = \{v|v \in \Gamma_{out}(v1) \wedge v1 \in V_{level1}\}$;
10: ......
11: Find the level-n node sets $V_{leveln} = \{v|v \in \Gamma_{out}(v(n-1)) \wedge v(n-1) \in V_{level(n-1)}\}$;
12: Return The level of each node in $nodes$
```

Rule 3: (**Hierarchy Rule.**) It is impossible for node x at level l to have an edge to a node y at level k iff l is larger than k, i.e., $e(x, y)$ must not exist if $l > k$.

Algorithm 3 Computation of Target to Source Nodes Similarity

```
1: ComputTargetToSourceSimilarity($target$,$source$)
2: Input:
3: $target$: Target node
4: $source$: Source node
5: Output: Similarity Score $sim$ of $target$ to $source$
6: Method:
7: Let $sim$ = 0;
8: Get the out-neighbors $outs$ of $target$;
9: for all $node \in outs$ do
10:     if $node$ is a source node then
11:         $sim$+=codeSim[$node$][$source$];
12:     else
13:         $sim$+=ComputTargetToSourceSimilarity($node$,$source$);
14:     end if
15: end for
16: if $|outs| \neq 0$ then
17:     Set $sim$=$sim/|outs|$;
18: end if
19: Return Similarity Score $sim$ of $target$ to $source$;
```

3) Code Similarity: Next, we want to compute the similarity between two nodes in a code graph. The code similarity between two nodes v and v' is given in Definition 6.

Definition 6: (**Code Similarity.**) Given a code graph $G(V, E)$, and two nodes v and v', if $e(v, v')$ exists, their similarity score is 1. However, if $e(v, v')$ does not exist, we define the code similarity score between nodes v and v' as the the ratio of the number of common out-neighbors and the number of out-neighbors of either node v or v', which is similar to the definition of Jaccard Index (JI) [4]. Formally, we define code similarity of two nodes v and v' as follows:

$$code_{sim}(v, v') = \{ \begin{array}{ll} 1, & if\ e(v, v') = 1 \\ \frac{|\Gamma_{out}(v) \cap \Gamma_{out}(v')|}{|\Gamma_{out}(v) \cup \Gamma_{out}(v')|}, & if\ e(v, v') \neq 1 \end{array} \quad (3)$$

4) Target to Source Nodes Similarity: Algorithm 3 presents an algorithm to compute a similarity score between a target node and a source node. Let's denote their similarity score as sim which is initialized to 0 (Line 7). After initializing this score, the algorithm gets the out-neighbors $outs$ of the target node (Line 8). Next, for each node in $outs$, it checks the type of the node (Line 9). If the node is a source node, it adds sim by the code similarity score of this node to the input source node (Lines 10-11). If the node is another target node, then it recursively calls itself (Lines 12-13). The final similarity score is computed as sim divided by the number of out-neighbors the target node has (Lines 16-17).

TABLE I. STATISTICS OF COLLECTED BUILD SYSTEMS.

Projects	# Nodes	# Edges	#Potential Links	# Sparsity
Zlib	91	233	4,095	5.69%
putty	99	411	4,851	8.48%
vim	146	1,144	10,585	10.81%
APR	289	1,223	41,616	2.94%
Memcached	227	2,443	25,651	9.52%
Nginx	291	6,798	42,195	16.11%
Tengine	320	8,258	51,040	16.18%

5) Target to Target Similarity: Algorithm 4 presents an algorithm to compute the similarity between a target node $targeta$ to another target node $targetb$. We denote their similarity score as sim which is initialized to 0 (Line 7). Similar to the computation of target to source node similarity, for each out-neighbor of $targeta$, the algorithm checks the type of the node. If the node is a source node, it adds a value to sim by calling $ComputeTargetToSourceSimilarity$ method (Line 11). If the node is a target node, then it recursively calls itself (Line 13). The final similarity score is computed as sim divided by the number of out-neighbors the target node $targeta$ has (Lines 16-17).

Algorithm 4 Computation of Target to Target Nodes Similarity

```
1: ComputeTargetToTargetSimilarity($targeta$,$targetb$)
2: Input:
3: $targeta$: target node
4: $targetb$: target node
5: Output: Similarity Score $sim$ of $targeta$ to $targetb$
6: Method:
7: Let $sim$ = 0;
8: Get the out-neighbors $outs$ of $targeta$;
9: for all $node \in outs$ do
10:     if $node$ is a source node then
11:         $sim$+=ComputeTargetToSourceSimilarity($targetb$, $node$)
12:     else
13:         $sim$+=ComputeTargetToTargetSimilarity($node$,$targetb$);
14:     end if
15: end for
16: if $|outs| \neq 0$ then
17:     Set $sim$=$sim/|outs|$;
18: end if
19: Return Similarity Score $sim$ of $targeta$ to $targetb$;
```

IV. EXPERIMENTS AND RESULTS

We evaluate *BuildPredictor* on the collected datasets in Table I. The columns correspond to the project name (Projects), the number of nodes in build graph (# Nodes), the number of edges (links) in Build Graph (# Edges), the number of potential links (#Nodes(#Nodes-1)/2) (# Potential Links), and the degree of sparsity (#Nodes/# Links) (# Sparsity), respectively. The experimental environment is a Windows 7 64-bit, Intel(R) Xeon(R) 2.53GH server with 24GB RAM.

Experiment Setup. We randomly divide the actual links from each build configuration file of the 7 projects into 10 sets of roughly equal sizes. Nine sets are used to create a partial build configuration file which is used as input to our approach, while the remaining 1 set is used to evaluate the performance of our approach. The whole process repeats 10 times, and finally we compute the average performance across the ten iterations. Our approach is similar to ten-fold cross validation which is an effective way to avoid the overfitting, and is widely used in software engineering studies, c.f., [7], [8], [9], [10], [11], [12].

One feature of our dependency mining problem is the imbalanced data phenomenon, i.e., the number of actual de-

TABLE II. Top-n Precision Scores for *BuildPreditor* and 9 Link Prediction Algorithms Used by Xia et al. for the 7 Build Systems. Aver. = The Average Top-n Precision Scores Across the 7 Build Systems.

Algo.	**Zlib**	**putty**	**vim**	**APR**	**Mem.**	**Nginx**	**Tengine**	**Aver.**
Build.	57.00%	100%	97.44%	51.20%	57.10%	100%	99.9%	80.38%
CN	0	0	0	0	0	0	0	0
CS	0	0	0	0.10%	0	0	0	0.01%
JI	0	0	0	0	0	0	0	0
AA	0	0	0.10%	0	0.10%	0	0	0.03%
RA	0	0	0.10%	0	0	0	0	0.01%
LHN1	0	0	0.10%	0	0	0	0	0.01%
PA	2.50%	0	0.10%	0.30%	9.20%	0	0	1.73%
Katz	0	0	0.30%	0	0	0	0	0.04%
LP	0	0	0	0	0	0	0	0

pendencies is small compared to the total number of possible dependencies. For example, in **APR** project, the number of nodes is 289, and thus the number of possible dependencies is $\frac{289*288}{2} = 41,616$, but the number of actual dependencies is 1,223, which is only $\frac{1,223}{41,616} = 2.94\%$ of the number of possible dependencies. Even for a denser dataset **Tengine**, the number of actual dependencies only takes $\frac{8,258}{51,040} = 16.18\%$ of the number of possible dependencies.

To evaluate the performance of our approach, we choose top-n precision and area under the ROC curve (AUC) values [4] as the evaluation metrics. They are defined in Definitions 7 & 8.

Definition 7: (**Top-n Precision.**) Consider a ranked list of top-n missed dependencies where N_r of them are actual missed dependencies. Then top-n precision is N_r/n. Clearly, a higher top-n precision means a higher prediction accuracy.

Definition 8: (**AUC Value.**) Given a dependency graph $G(V, E)$, we denote the nonexisting links in the graph as $U-E$, and denote the actual missed links as E^T. We randomly select a link from E^T, and compute its similarity score $score_T$; We also randomly select a link from $U - E$, and compute its similarity score $score_{none}$. We repeat this procedure with m times, and record the number of times $score_T > score_{none}$ (denoted as m_1), and the number of times $score_T = score_{none}$ (denote as m_2). The AUC value is computed as:

$$AUC = \frac{m_1 + 0.5 \times m_2}{m} \quad (4)$$

AUC measures the likelihood that an actual missing dependency is given a higher similarity score than a false positive (i.e., a non-existent dependency). If all the similarity scores are generated from an independent and identical distribution, the AUC should between 0.5 and 1 [4]. The higher an AUC value is, the better performance an algorithm achieves. Moreover, if AUC is below 0.5, it means this algorithm is even worse than random guess.

In the evaluation, we set the parameters for the computation of top-n precision and AUC as follows:

- For Zlib and putty, since the number of nodes and links are small, we set n to 20. For the remaining projects, we set n to 100.
- The parameter m for AUC value computation is set as 10,000;

Experiment Results. We compare *BuildPredictor* with the nine link prediction algorithms used by Xia et al. – common neighbors (CN), cosine similarity (CS), Jaccard Index (JI), Adamic-Adar (AA), Resource Allocation (RA), Leicht-Holme-Newman (LHN1), preferential attachment (PA), Katz, and local path index (LP) [3].

TABLE III. AUC Scores for *BuildPredictor* and 9 Link Prediction Algorithms Used by Xia et al. for the 7 Build Systems. Aver. = Average AUC Scores Across the 7 Build Systems

Algo.	**Zlib**	**putty**	**vim**	**APR**	**Mem.**	**Nginx**	**Tengine**	**Aver.**
Build.	0.8550	0.8603	0.8639	0.8716	0.6136	0.9235	0.9703	0.8512
CN	0.3876	0.3772	0.3958	0.3853	0.2974	0.3384	0.3376	0.3599
CS	0.3886	0.3781	0.3896	0.3862	0.2962	0.3398	0.3378	0.3595
JI	0.3870	0.3775	0.3915	0.3857	0.2984	0.3370	0.3380	0.3593
AA	0.3870	0.3784	0.3944	0.3865	0.3014	0.3385	0.3376	0.3605
RA	0.3873	0.3769	0.3933	0.3853	0.3076	0.3379	0.3380	0.3609
LHN1	0.3875	0.3777	0.3958	0.3859	0.2943	0.3393	0.3382	0.3598
PA	0.7169	0.8264	0.8503	0.8654	0.8728	0.8485	0.8522	0.8332
Katz	0.5865	0.5954	0.6445	0.7134	0.5020	0.6462	0.6466	0.6192
LP	0.6137	0.6447	0.6496	0.7007	0.5058	0.7071	0.7051	0.6467

Table II presents the results of the algorithms' top-n precision scores for the 7 build systems. The results show that *BuildPredictor* achieves much better performance than all the other algorithms. The top-n precision scores for *BuildPredictor* vary from 51.2% to 100%, and the average top-n precision across the 7 build systems is 80.38%. For the link prediction algorithms, only PA could predict some missed dependencies correctly in its top-n returned results; its average top-n precision is only 1.73%, which is very poor as compared to the performance of *BuildPredictor*. Table III presents the results of the algorithms' AUC scores for the 7 build systems. The AUC values of CN, CS, JI, AA, RA, and LHN1 are extremely low; they are lower than 0.5. And the AUC values for *BuildPredictor*, PA, Katz and LP are much better, which are more than 0.5. Among the 10 algorithms, we notice that our *BuildPredictor* achieves the best performance in terms of AUC; its AUC scores vary from 0.6136 to 0.9703, and its average AUC across the 7 build systems is 0.8512. PA achieves the second best performance; its AUC scores vary from 0.7169 to 0.8268, and its average AUC is 0.8332.

Top-n precision measures the accuracy of the top few recommendations made by a dependency mining approach, while AUC measures the overall accuracy of the entire recommendations. Since developers are likely to investigate only top few recommendations and would likely not use the tool if the top few recommendations are bad (c.f., [5]), achieving high top-n precision is more important than high AUC.

Threats to Validity. Threats to internal validity relate to errors in our experiments. We have double checked our experiments, still there could be errors that we did not notice. We reuse the datasets provided by Xia et al. which are previously used to evaluate the 9 link prediction algorithms [3]. Threats to external validity relate to the generalizability of our results. We have evaluated *BuildPredictor* on 7 build configuration files from various projects. In the future, we plan to reduce this threat further by analyzing more build configuration files from other projects, including commercial and open source projects. Threats to construct validity refer to the suitability of our evaluation metrics. We use top-n precision and AUC as the main evaluation metrics which are also used by many past software engineering and data mining studies to evaluate the effectiveness of a prediction technique [4], [13], [14]. Thus, we believe there is little threat to construct validity.

V. Related Work

Dependency Mining. The most related work to our paper is the work by Xia et al. [3], which proposes the dependency mining problem, which is the task to predict missed dependencies in build configuration files. Xia et al. leverage 9 state-of-the-art link prediction algorithms to predict the missed dependencies, and they conclude that preferential attachment (PA) achieves the best performance. It's AUC values vary from 0.71 - 0.87. Our study extends their study: we consider both the dependency graph extracted from build configuration files, and code graph extracted from source code files, and propose a more accurate method to predict the missed dependencies.

Studies on Build System Maintenance. MAKAO is a visualization and smell detection tool for make-based build system [6]. MAKAO generates a dependency graph from a Makefile, and based on this, it supports various functionalities such as querying build-related data, and viewing the build architecture from different aspects. McIntosh et al. investigate coupling between source code and build system changes in ten software projects to measure the cost and effort of build maintenance [17]. Suvorov et al. perform an empirical study to investigate the migration of build systems performed in two open source projects: Linux Kernel and KDE [18]. Neitsch et al. study issues in build systems for multiple programming languages, and explore the root cause of the issues [19]. Tu and Godfrey investigate the characteristics and benefits of build-time software architecture on GCC, Perl, and JNI [20]. Xia el al. perform an empirical study on bugs in 4 software build systems – make, ant, cmake, and qmake [21]. Zhao et al. investigate bugs related to software build process, and they find that build process related bugs take approximately double the amount of time to be fixed than other bugs [22].

VI. Conclusion and Future Work

In this paper, we propose *BuildPredictor* to automatically predict the missed dependencies in build configuration files. *BuildPredictor* considers both dependency graph extracted from a build configuration file and code graph extracted from source code, and recommends a list of candidate dependencies that are likely to be missed. We evaluate the effectiveness of *BuildPredictor* on 7 build configuration files from various open source software projects. We compare the performance of *BuildPredictor* against the performance of 9 link prediction algorithms used by Xia et al. to predict missed dependencies. The experiment results show that on average our *BuildPredictor* achieves the best performance; it achieves an average top-n precision and AUC score of 80.38% and 0.8512 respectively across the 7 projects, which are higher than the results of the 9 link prediction algorithms.

In the future, we plan to investigate more build configuration files from other projects to further evaluate the effectiveness of *BuildPredictor*. We also plan to improve the accuracy (e.g., top-n precision and AUC values) of *BuildPredictor* further.

Acknowledgment. This research is sponsored in part by NSFC Program (No.61103032) and National Key Technology R&D Program of the Ministry of Science and Technology of China (2014BAH24F02).

References

[1] P. Smith, *Software Build Systems: Principles and Experience*. Addison-Wesley Professional, 2011.

[2] G. Epperly, "Software in the doe: The hidden overhead of the build," 2002.

[3] X. Xia, D. Lo, X. Wang, and B. Zhou, "Build system analysis with link prediction," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014.

[4] L. Lü and T. Zhou, "Link prediction in complex networks: A survey," *Physica A: Statistical Mechanics and its Applications*, vol. 390, no. 6, pp. 1150–1170, 2011.

[5] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 199–209.

[6] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*. IEEE, 2007, pp. 114–123.

[7] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto, "Predicting re-opened bugs: A case study on the eclipse project," in *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 2010, pp. 249–258.

[8] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *CSMR-WCRE*, 2014.

[9] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proceedings of the Tenth International Workshop on Mining Software Repositories*. IEEE Press, 2013, pp. 287–296.

[10] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 386–396.

[11] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," in *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. IEEE, 2008, pp. 346–355.

[12] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 182–191.

[13] A. Menon and C. Elkan, "Link prediction via matrix factorization," *Machine Learning and Knowledge Discovery in Databases*, pp. 437–452, 2011.

[14] T. Zhou, L. Lü, and Y. Zhang, "Predicting missing links via local information," *The European Physical Journal B-Condensed Matter and Complex Systems*, vol. 71, no. 4, pp. 623–630, 2009.

[15] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build code analysis with symbolic evaluation," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 650–660.

[16] ——, "Symake: a build code analysis and refactoring tool for makefiles," in *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. IEEE, 2012, pp. 366–369.

[17] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd international conference on software engineering*. ACM, 2011, pp. 141–150.

[18] R. Suvorov, M. Nagappan, A. E. Hassan, Y. Zou, and B. Adams, "An empirical study of build system migrations in practice: Case studies on kde and the linux kernel," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 160–169.

[19] A. Neitsch, K. Wong, and M. W. Godfrey, "Build system issues in multilanguage software," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 140–149.

[20] Q. Tu and M. W. Godfrey, "The build-time software architecture view," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. IEEE Computer Society, 2001, p. 398.

[21] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," in *Quality Software (QSIC), 2013 13th International Conference on*. IEEE, 2013, pp. 200–203.

[22] X. Zhao, X. Xia, P. Kochhar, D. Lo, and S. Li, "An empirical study of bugs in build process," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014.