2014 IEEE International Conference on Software Maintenance and Evolution

# Fault Localization for Make-Based Build Crashes

Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen
Electrical and Computer Engineering Department, Iowa State University, USA
Email: {jafar,hungnv,tien}@iastate.edu

***Abstract*—In large-scale software projects, build code has a high level of complexity, churn rate, and defect proneness. While it is desirable to have automated tools to help developers in localizing faults in build code, it is challenging to build such tools due to the dynamic nature of build code. Existing automatic fault localization methods focus on traditional code and none of them has such support for build code. This paper introduces MkFault, a novel automatic tool/method to localize faults in build code that cause run-time build failures. Given a test case that causes a run-time crash in the execution of a Makefile, it returns a ranked list of statements in the Makefile with their suspiciousness scores. MkFault records the evaluation traces from Make code to identify the corresponding concrete build rules and the execution traces of those rules. It then uses those traces and its novel Bayesian-like rating algorithm to give suspiciousness scores to the original statements in the Makefile. Our empirical evaluation on real faults in several open-source projects has shown that MkFault can achieve high accuracy and help reduce a large percentage of the lines of code that developers need to examine.**

## I. Introduction and Motivation

In a software project, building is a crucial process which compiles, links, integrates, and converts source code, libraries, and resources into independent deliverables and executable files. To perform software building, developers write build files in a scripting language to instruct a build tool (e.g., GNU Make [13] and Ant [4]) to execute the build commands following the rules specified in the build scripts. In large projects, build code has a high level of complexity, churn rate, and defect proneness [21], [15]. Adams *et al.* [2] found that build code in Linux co-evolves with source code with increasing complexity. McIntosh *et al.* [21] reported that build code continually evolves more complex and defect-prone due to its high churn rate. Hochstein and Jiao [15] found that 11%–47% of test failures are build-related. While there exist automated approaches to help developers in localizing faults in traditional code, and in detecting smells in build code (e.g., MAKAO [3], SYMake [26]), or debugging Makefiles (e.g., ReMake [23]), none of them supports localizing a fault causing a build crash in large and complex build code.

Figure 1 shows a Makefile specifying the rules to build a program from the corresponding source code in Java or C. Make processes a Makefile in two distinct phases:

1) *Evaluation phase*: Make first processes the Makefile to produce concrete rules and construct a *concrete dependency graph* (CDG). A rule in a CDG specifies a dependency between *prerequisites* and *targets*, and a *recipe* (i.e. shell commands) to build the targets from their prerequisites. Figure 2 displays the rules after the evaluation phase on myMakefile, provided that the environment contains Java source files.

2) *Execution phase*: With its internal representation of concrete rules, Make executes the required recipes to generate targets from their prerequisites. For example, assume that Make is invoked with the command 'make program.jar', the rule for the target program.jar is executed (lines 10-12, Figure 2), which causes the rules for Main.class and Util.class to be executed.

```
1  WSPACE := wp
2  SRCFILES := $(foreach dir, $(WSPACE), $(wildcard $(dir)/*.java))
3
4  ifeq ($(strip $(SRCFILES)),)
5      SRCFILES := $(foreach dir, $(WSPACE), $(wildcard $(dir)/*.c))
6      CMPFILES := $(SRCFILES:.c=.o)
7      ext :=
8      build = link /out:$@ $^
9  else
10     CMPFILES := $(SRCFILES:.java=.class)
11     ext :=.jar
12     build = jar cf $@ $(CMPFILES) $(WSPACE)/lib.jar
13 endif
14
15 %.class: %.java
16     javac -classpath $(WSPACE) $^
17
18 %.o: %.c
19     $(CC) -c $(CFLAGS) $^ -o $@
20
21 cleanCmd = for /f "usebackq" %%i in (`dir $(WSPACE) /s /b ^| findstr /vi
       ". java$$ .c$$"`) do del /q %%i
22
23 clean:
24     $(cleanCmd)
25
26 program$(ext): $(CMPFILES)
27     if exist $@ ( \
28         del /f $@ )
29     $(build)
30
31 all : clean program$(ext)
```

Fig. 1. myMakefile: An example of a Makefile

```
1  wp/Main.class: wp/Main.java
2      javac -classpath wp wp/Main.java
3
4  wp/Util.class: wp/Util.java
5      javac -classpath wp wp/Util.java
6
7  clean:
8      for /f "usebackq" %%i in (`dir wp /s /b ^| findstr /vi ". java$ .c$
           "`) do del /q %%i
9
10 program.jar: wp/Main.class wp/Util.class
11     if exist program.jar (del /f program.jar)
12     jar cf program.jar wp/Main.class wp/Util.class wp/lib.jar
13
14 all : clean program.jar
```

Fig. 2. Internal concrete rules after evaluation for Java (invisible from users)

**Errors.** Figure 3 illustrates an error that occurred during the execution of 'make all'. According to the rule for target all (line 14, Figure 2), the rule for clean should be processed first, followed by the rule for program.jar. As seen in the error message, the failure happened when the recipe to create the

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.87

```
1 wp/lib.jar : no such file or directory
2 make: *** [program.jar] Error 1
```

Fig. 3. Error in the execution phase on myMakefile

TABLE I. BUILD CODE COMPLEXITY

| Systems | Build Files | SLOCs | Vars | Rules | Paths | Max Included Files |
|---|---|---|---|---|---|---|
| SCST | 49 | 1,786 | 876 | 112 | 154 | 10 |
| Linux2.6-net | 67 | 4,020 | 3,425 | 134 | 536 | 20 |
| Gcc | 68 | 5,350 | 1,980 | 804 | 75 | 5 |
| Minix | 95 | 2,374 | 632 | 121 | 95 | 95 |
| Linux2.6-sound | 98 | 1,255 | 973 | 135 | 98 | 10 |
| Firefox-Gecko | 156 | 6,374 | 1,991 | 2,635 | 621 | 130 |
| Thunderbird | 232 | 12,950 | 2,655 | 2,541 | 235 | 210 |

target program.jar was executed (lines 10-12, Figure 2), and it was due to a missing file (wp/lib.jar).

Given that, it is not clear for a developer where the root cause of the error is in *the original* myMakefile. The error message is reported for a *faulty concrete rule* in Figure 2, which is invisible from a developer. In this case, the error occurred not because the rule to create program.jar itself is incorrect, but because a *previously executed rule (i.e., the rule for clean) mistakenly deleted the library file* wp/lib.jar (line 8, Figure 2). The *actual root cause* of the failure in the original myMakefile is at line 21 of Figure 1, where the faulty recipe was assigned to the variable cleanCmd and was evaluated on line 24, Figure 1, into the concrete recipe on line 8, Figure 2.

To learn more about the complexity of Make code, we have performed a preliminary study on seven open-source projects that use Make. As seen in Table I, developers must work on a significant amount of build code, e.g., with 232 build files and 13K lines of build code in Thunderbird. Importantly, in a project, there are up to 2,635 build rules with up to 621 different execution paths to build deliverables. Top-level Makefiles also include up to 210 other Makefiles. Thus, it is desirable to have automated tools to localize defects in Makefiles.

**Challenges**. It is challenging for a tool to localize such a fault:

1) The analysis for build dependencies among files is not trivial due to the two-phase dynamic nature of Make. To localize a fault, a tool needs to analyze not only the faulty concrete rules but also their originating code in Makefiles.

2) A faulty rule may be unexposed and may manifest itself as a crash *only during the execution of another rule* after it. Thus, it is not self-evident what concrete rule is faulty and responsible for a given build crash when running Make.

3) Current fault localization techniques on a regular program often leverage *multiple* passing and failing test cases for it. However, creating test oracles for a Makefile is challenging due to the task of determining the expected output for a run on the Makefile. Thus, to localize a build fault, a tool has *only one* failing test case that causes the crash.

In this paper, we introduce MkFault, a tool to help localize faults in build code written in Make that lead to run-time crashes in the execution phase. Given a test case that causes a run-time crash during the execution of a build script (a *Makefile*), MkFault returns a ranked list of statements in the Makefile with suspiciousness scores. We instrument MkFault's code into GNU Make to record the code locations in the Makefile that were used to generate the concrete build rules when the test case was run. That is, for each character in a concrete build rule (target, prerequisites, or recipe), MkFault is able to map it to the sequence of corresponding statements in the Makefile, which we call *an evaluation trace*. Our instrumented code also records the *execution trace* via the concrete build rules during the execution phase, as well as the crash point (i.e., the concrete rule where the execution stops). Given the crash point and the execution and evaluation traces for a single test case, our novel rating algorithm will compute the suspiciousness scores in the Makefile via a *Bayesian-like probability computation*. Our empirical evaluation on real faults in several open-source projects showed that MkFault achieves high accuracy (up to 88.6% for top-5 accuracy) in localizing build faults causing crashes, and reduces 81.7%–95.7% of the lines that developers need to examine. Our key contributions include

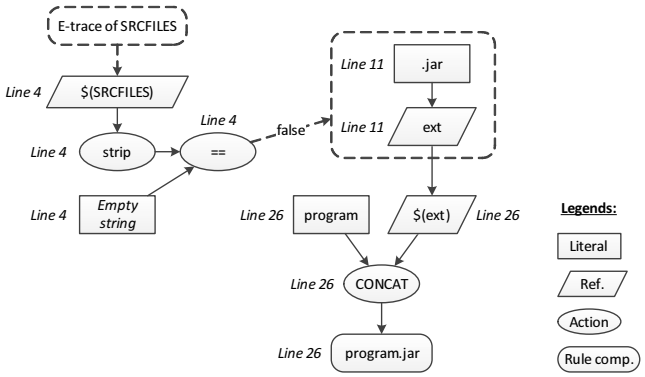


Fig. 4. The E-trace of the target program.jar on line 26 of Figure 1

1. MkFault, an automatic fault localization method for localizing build faults in Makefiles that can cause build crashes,

2. An empirical evaluation on MkFault's accuracy and usefulness in localizing build code errors.

## II. DYNAMIC INSTRUMENTATION

This section describes MkFault's instrumentation into GNU Make to build the *evaluation trace* and *execution trace* of the concrete build rules. These traces will be used in our rating algorithm to localize faults on a Makefile (Section III).

### A. Evaluation Trace of Concrete Build Rules

The concrete build rules in the CDG (including their targets, prerequisites, and recipes) are often computed, composed, and manipulated at different code locations in a Makefile. For example, the target program.jar on line 10 of Figure 2 is generated from the expression program$(ext) on line 26 of Figure 1, in which 'program' is a string literal and $(ext) is a variable which is assigned with the string '.jar' at line 11 of Figure 1.

To capture that process, we develop *E-trace*, a model for tracing the generation of the concrete build rules. Different node types are designed to model the elements in each step.

*Definition 1 (E-trace): An E-trace (evaluation trace) is a labeled, directed, and acyclic graph representing how the*

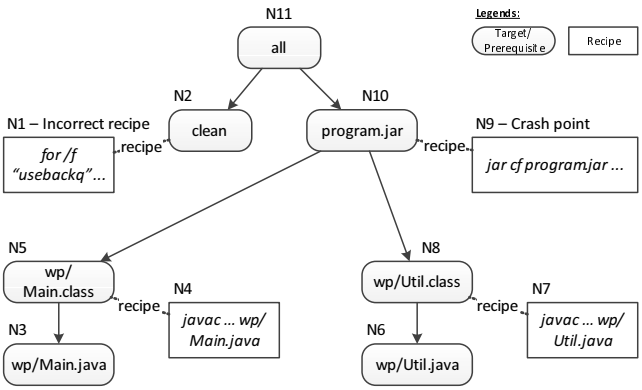


Fig. 5. The CDG and the execution trace of myMakefile with the crash at N9

*concrete build rules in a CDG are computed and manipulated through Makefile's program elements. A node refers to an expression at a code location in the Makefile. The edges represent the evaluation flows among those expressions.*

Figure 4 shows the E-trace that produces the target program.jar resulted from the evaluation of myMakefile (Figure 1). The left-hand side of Figure 4 shows the computation steps from lines 1-4 of Figure 1, which lead to the comparison operator of the ifeq statement (line 4). Since the current evaluation is for a configuration with Java, it continues with the else branch (line 9), whose E-trace is partially shown in the dotted rectangle. The target program.jar is concatenated from the string 'program' (modeled by a Literal node) and the variable $(ext) (modeled by a Reference node, line 26), whose value is obtained via a variable assignment (line 11). All code locations on the E-trace for a rule component contribute to the creation of that component and are considered in localizing faults.

### B. Execution Trace of Concrete Build Rules

In the execution phase, Make processes the concrete build rules in the CDG and executes their recipes. For a given rule $r$ and a target $t$ specified by $r$, Make first processes the rules to create the prerequisites of $t$ and then executes the recipe given by $r$ to create target $t$. If the target is already updated, Make will skip processing the prerequisites' rules and the recipe. MkFault instruments GNU Make to follow this process and records the *execution trace* of the rules and recipes executed by Make.

Figure 5 shows the CDG of myMakefile for Java and the execution trace to the crash point when the recipe to create program.jar was being executed. *The execution trace, which is built from the CDG, includes not only the executed recipe nodes, but also the target/prerequisite nodes* to indicate that resources are created after recipes are executed. In the trace, for a given rule, the prerequisites' nodes are placed before the recipe's node, followed by the target's node. The order among the prerequisite nodes is the same as their appearance order in the rule. For instance, for the target N5, in the trace, the prerequisite N3 is placed first, then the recipe N4, and finally N5. In this example, the execution trace in MkFault is N1–N9.

When a crash takes place, the target of the current rule cannot be produced (e.g., the target program.jar at N10 cannot be created due to the crash at N9). Thus, after the crash, the execution is said to be in an *incorrect execution state*.

*Definition 2 (Execution state): The execution state (or state for short) during the execution phase is the set of files/resources used in the execution and their contents.*

In general, an incorrect state at a given rule can be caused by a fault in the current rule, or propagated from a fault in the rules for its prerequisites or its *preceding target*.

*Definition 3 (Preceding target): A preceding target $p$ (or precedent for short) of a target $t$ in an execution is the last target that is executed before $t$, except those that are descendants of $t$ in the CDG.*

In Figure 5, the target clean is the precedent of the target progam.jar. A target's prerequisites, precedent and their execution states will be used in the fault localization step.

## III. Fault Localization on Makefiles

Since Make processes a Makefile in two phases, when a crash occurs, MkFault localizes the fault in two steps corresponding to those two phases: The first step aims to identify the faulty rule in the CDG that leads to the crash, while the second step pinpoints the location in the original Makefile that is responsible for generating that faulty rule.

### A. Computing Suspiciousness Scores on Concrete Dependency Graph

In the first step, we distribute fault probabilities over different nodes in the CDG. This process starts with the concrete rule where the crash occurs with its probability of 100% for being in an incorrect state. This incorrect state at the crash point can be caused by the concrete rule at the crash point being incorrect itself, or by the incorrect state of one of the rules for the preceding target or prerequisites. Thus, the probability is distributed among those three sources of inaccuracy. Then, the distribution of probabilities continues for the nodes at the preceding target and prerequisites. These probabilities decrease multiplicatively with the number of nodes as the process descends in the CDG further away from the crashing node. The result of this step is the fault probabilities for all concrete rules.

**Example.** Let us illustrate our inference via an example for myMakefile (Figure 6). First we define the following random variables to represent an execution state and address the three causes of an incorrect execution state.

*Definition 4 (Correct/incorrect state): Let $S(r)$ be a random variable that represents the event that the execution after executing a rule $r$ is in a correct state ($S(r)$ = True) or in an incorrect state ($S(r)$ = False).*

*Definition 5 (Correct/incorrect rule): Let $R(r)$ be a random variable that represents the event that a rule $r$ is correct ($R(r)$ = True) or incorrect ($R(r)$= False).*

*Definition 6 (Exclusively (in)correct state): Let $X(r)$ be a random variable that represents the event that the state after executing the rule $r$ is correct ($X(r)$ = True) or incorrect ($X(r)$ = False), assuming that the state after executing the preceding rule of $r$ is correct ($S(Prec(r))$ = True). If $r$ does not have a*

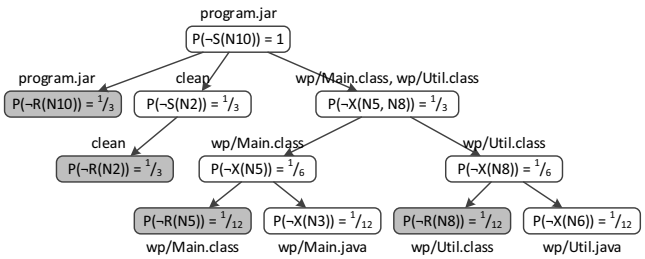


Fig. 6. Computing probabilities for myMakefile's concrete build rules (based on the CDG in Figure 5)

*preceding target, then $X(r) = S(r)$. $X$ can be applied to a set of rules.*

In Figure 6, the probability $P(\neg S(\text{program.jar})) = 1$ since program.jar is the crashed rule. This probability is distributed among the three possible causes of the crash, namely $P(\neg R(\text{program.jar}))$, $P(\neg S(\text{clean}))$, and $P(\neg X(\text{wp/Main.class, wp/Util.class}))$ (with each value equal to $\frac{1}{3}$). The last probability is divided equally between the two prerequisites of program.jar: $P(\neg X(\text{wp/Main.class}))$=$P(\neg X(\text{wp/Util.class}))$=$\frac{1}{6}$. These values are then used to compute other probabilities in the same way. For example, since the target clean does not have a target or a prerequisite, $P(\neg R(\text{clean})) = P(\neg S(\text{clean})) = \frac{1}{3}$. The algorithm finishes with the values of all $P(\neg R(r))$ being computed (highlighted nodes in Figure 6), indicating the probability that a given rule is incorrect. Since the target all is not found in the execution trace (hence the causal graph), $P(\neg R(\text{all})) = 0$. As seen, the rule clean which contains an incorrect recipe causing the crash is ranked at the top-2 suspicious rules.

### *B.* Localizing Faults on Makefiles from Fault Probabilities on CDG

Given the fault probability of a concrete rule, MkFault distributes that probability over all the code locations in the Makefile that are responsible for generating the concrete rule based on the E-trace of that rule. Due to Make's dynamism, one code location may contribute to the generation of different concrete rules (e.g., the code location containing a variable definition is involved with the rules in which the variable is used). Thus, to compute the probability $P(F(l))$ that a given code location $l$ contains a fault, MkFault performs a summation over the joint probabilities of $l$ and all the concrete rules in the CDG that are entirely or partly generated from $l$. Specifically, let $E(l)$ be the set of rules whose E-traces contain the code location $l$, $P(F(l))$ can be computed as follows.

$$P(F(l)) \quad = \quad \sum_{r \in E(l)} \frac{P(\neg R(r))}{|\text{E-trace}(r)|}$$

The final result is a ranked list of locations in the Makefile, each with an associated probability (i.e., suspiciousness score) indicating the likelihood that it has a fault.

**Example.** Table II illustrates the computation of $P(F(l))$ for all code locations in myMakefile (Figure 1). ($P(\neg R(r))$ for all the rules were computed from Figure 6.) As seen, in this example, the location containing the root cause of the crash (line 21) is ranked among the top 2-4 locations.

TABLE II. POBABILITIES FOR MYMAKEFILE'S CODE LOCATIONS

(a) $P(\neg R(r))$ and E-trace of a rule $r$

| Rule $r$ | E-trace($r$) (Lines) | $P(\neg R(r))$ | \|E-trace($r$)\| |
|---|---|---|---|
| program.jar | 26,27,28,29,11,4,2,1,12,10 | $^1/_3$ | 10 |
| clean | 23, 24, 21, 1 | $^1/_3$ | 4 |
| Main.class | 15, 16, 10, 4, 2, 1 | $^1/_{12}$ | 6 |
| Util.class | 15, 16, 10, 4, 2, 1 | $^1/_{12}$ | 6 |
| all | 31, 11, 4, 2, 1 | 0 | 5 |

(b) Computing $P(F(l))$ for a location $l$

| Line $l$ | Sus. score ($P(F(l))$) | Rank |
|---|---|---|
| 1 | 0.144 | 1 |
| 21, 23, 24 | 0.083 | 2-4 |
| 2, 4, 10 | 0.061 | 5-7 |
| 11, 12, 26, 27, 28, 29 | 0.033 | 8-13 |
| 15, 16 | 0.028 | 14-15 |
| Others | 0 | - |

TABLE III. SUBJECT SYSTEMS AND THEIR COMPLEXITY

| System | Bugs | MF | LOC | Vars | Rules | CDG Nodes | CDG Edges | E-trace/ Node | ExTrace |
|---|---|---|---|---|---|---|---|---|---|
| S1 | 13 | 14 | 2166 | 186 | 264 | 402 | 799 | 7 | 287 |
| S2 | 18 | 1 | 437 | 15 | 23 | 404 | 636 | 5 | 470 |
| S3 | 17 | 2 | 302 | 25 | 19 | 73 | 215 | 11 | 66 |
| Tot/$\overline{\text{Avg}}$ | 48 | 17 | 2905 | 226 | 306 | $\overline{293}$ | $\overline{550}$ | $\overline{8}$ | $\overline{274}$ |

## IV. EMPIRICAL EVALUATION

We conducted an empirical experiment to evaluate MkFault's accuracy and how it helps reduce efforts in localizing Make build code faults. We first collected several open-source subject projects from sourceforge.net that use Make as their build language and have a long development history. They include Dream Toolbox[12] (S1), GMod [14] (S2), and X10 [27] (S3). We built a tool to select for each subject project the revisions that have at least one modified Makefile. We randomly selected one revision as a starting point. Then, we manually examined the changes to the Makefiles as well as the commit logs to determine if they were the bug fixes to build crashes in those Makefiles. If such a build error was found, we compared the current revision (fixed one) with the previous revision (buggy one), and used the fixing change location in the buggy revision as the root cause of the error. We collected that root cause location and that buggy Makefile. We skipped non-crashing faults, and the faults involving multiple fixing locations. Then, we continued the process for the next revisions until we had 10-20 faults for each project and used them as an oracle.

In Table III, column Bugs shows the total number of bugs collected for a project. For each revision containing a bug, we computed eight complexity metrics and took the average numbers across those revisions. The next four columns show the complexity of the Makefiles: the number of involved Makefiles, the number of LOC in those Makefiles and the number of program elements including variables and rules. The last four columns show complexity metrics collected in an execution that results in a crash including the size of the generated CDG, the average length of E-trace per CDG node, and the length of the execution trace. As seen, to handle those complex buggy Makefiles, one would need to have tool support as in MkFault.

TABLE IV. MKFAULT'S FAULT LOCALIZATION RESULTS

| Sys. | TrLines | $E_{low}$ | $E_{high}$ | $E_{mean}$ | Top-1 | Top-5 | Top-10 |
|---|---|---|---|---|---|---|---|
| S1 | 243 | 88.2% | 98.9% | 93.6% | 18.3-83.3% | 75-100% | 83.3-100% |
| S2 | 26 | 80.2% | 93.0% | 86.6% | 16.7-72.2% | 27.8-83.3% | 27.8-94.4% |
| S3 | 45 | 76.8% | 95.3% | 86.0% | 15.0-76.5% | 15.9-82.4% | 17.6-94.1% |
| Avg. | **104.7** | **81.7%** | **95.7%** | **88.7%** | **16.7-77.3%** | **39.6-88.6%** | **42.9-96.2%** |

For each bug in the oracle, we ran MkFault on the buggy Makefile (and the included ones) to produce the list of suspicious lines in the Makefiles with their suspiciousness scores. We measured MkFault's performance by its *effectiveness score* and *top-$n$ accuracy*. The effectiveness score $E$ is defined as:

$$E = 1 - \frac{\text{Rank(fault)}}{\text{TrLines}}$$

where Rank(fault) is the rank of the faulty line in MkFault's ranked list and TrLines is the number of lines that are involved in the *evaluation and execution traces* of the Makefile (i.e., the number of lines that a developer would need to inspect using a debugger when detecting the fault). That is, the effectiveness score $E$ is the percentage of lines that *need not* be inspected by the developer by using MkFault's results. This measure has been used in previous fault localization studies (e.g., [9]). A higher effectiveness score indicates more effort being saved in fault localization. If the faulty line has the same suspiciousness score with other lines, its rank can vary from the smallest to the largest ranking number for that set of lines (called $S_{fault}$ set). For example, with the following suspiciousness scores assigned to lines L1 to L5: L1=0.7, L2=0.3, L3=0.5, L4=0.9, L5=0.7, and assuming L1 is faulty, then $S_{fault}$={L1, L5} and L1 can rank either second or third out of the five lines. To address such cases, we compute the effectiveness score as both $E_{high}$ and $E_{low}$ for the highest and lowest ranks of $S_{fault}$ (for the previous example, $E_{high} = 60\%$ and $E_{low} = 40\%$). We also recorded the effectiveness score for the mean rank ($E_{mean}$).

For *top-$n$ accuracy*, we count the number of times (or hits) that the faulty line is ranked among the top $n$ of the ranked list returned by MkFault. Top-$n$ accuracy is measured by the ratio of the number of hits over the total localization cases.

**Results.** Table IV shows the results. For top-$n$ accuracy, each cell has two numbers corresponding to the cases where the faulty line is to be ranked at the highest or the lowest among the set of lines with the same score. As seen, $E_{high}$ is in 93.0–98.9%, and $E_{low}$ is in 76.8–88.2%. On average, $E_{high}$ is 95.7%, $E_{low}$ is 81.7%, and its mean $E_{mean}$ is 88.7%. Thus, MkFault has high effectiveness and could help save debugging effort of 88.7% on average. Also, MkFault can achieve high accuracy. In up to 77.3% of the cases, a single recommended location contains the fault. In up to 88.6% of the cases, one could find the fault in the first 5 recommended lines. The variance in top-n accuracy (e.g., 39.6–88.6% for top-5) is due to the fact that the faulty line often shares the same suspiciousness score with other lines and their rankings range from less than the n-th rank to more than the n-th rank (i.e., falling out of the top-n result). The complete results are on our project's website [22].

## V. CONCLUSION

Building is an important process in software development. This paper presents MkFault, a novel method to localize faults in build code that cause run-time build crashes. MkFault records the evaluation traces from Make code statements that produce the corresponding concrete build rules and the execution traces of those rules. It then uses those traces and its novel rating algorithm to give suspiciousness scores to the original statements in the Makefile. Our empirical evaluation on real faults has shown that MkFault is highly accurate and helps reduce the number of lines of code that need to be examined in fault localization.

## REFERENCES


[1] Action Game. http://sourceforge.net/projects/actiongame/.

[2] B. Adams, K. D. Schutter, H. Tromp, and W. De Meuter. The evolution of the linux build system. *Electronic Comm. of the EASST*, 2007.

[3] B. Adams, H. Tromp, K. De Schutter, W. De Meuter. Design recovery and maintenance of build systems. In ICSM'07. IEEE, 2007.

[4] Apache Ant User Manual. http://ant.apache.org/manual/.

[5] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, M. Ernst. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE TSE*, 36:474–494, July 2010.

[6] Blood Frontier. http://sourceforge.net/projects/bloodfrontier/

[7] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In ICSE '09, pages 34–44. IEEE CS, 2009.

[8] S. R. Clark, J. Cobb, G. M. Kapfhammer, J. A. Jones, M. J. Harrold. Localizing SQL faults in database applications. In ASE'11. IEEE, 2011.

[9] H. Cleve and A. Zeller. Locating causes of program failures. In ICSE'05, pages 342–351. ACM, 2005.

[10] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *ECOOP*, pages 528–550, 2005.

[11] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, and D. Weiss. Customization change impact analysis for erp professionals via program slicing. In ISSTA '08, pages 97–108. ACM, 2008.

[12] The DREAM Toolbox. http://sourceforge.net/projects/dreamtoolbox/

[13] S. I. Feldman. Make: A program for maintaining computer programs. *Software Practice*, 9:255–265, 1979.

[14] GMOD. http://sourceforge.net/projects/gmod/?source=directory.

[15] L. Hochstein and Y. Jiao. The cost of the build tax in scientific software. In ESEM '11. IEEE CS, 2011.

[16] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In ASE '05. ACM, 2005.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In PLDI '05, pages 15–26. ACM, 2005.

[18] S. Litvak, N. Dor, R. Bodik, N. Rinetzky, and M. Sagiv. Field-sensitive program dependence analysis. In FSE '10, pages 287–296. ACM, 2010.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In FSE '05, pages 286–295. ACM, 2005.

[20] S. Mani, V. S. Sinha, P. Dhoolia, and S. Sinha. Automated support for repairing input-model faults. In ASE '10, pages 195–204. ACM, 2010.

[21] S. McIntosh, B. Adams, T. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In ICSE '11. ACM, 2011.

[22] MkFault. http://home.engineering.iastate.edu/~jafar/mkfault/.

[23] Rocky Bernstein. Remake: GNU Make with comprehensible tracing and a debugger. http://bashdb.sourceforge.net/remake.

[24] G. Robles, J. M. Gonzalez-Barahona, and J. J. Merelo. Beyond source code: the importance of other artifacts in software development (a case study). *J. Syst. Softw.*, 79:1233–1248, September 2006.

[25] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweight fault-localization using multiple coverage types. In ICSE '09. IEEE, 2009.

[26] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. Build Code Analysis with Symbolic Evaluation. In ICSE '12. IEEE, 2012.

[27] X10. http://sourceforge.net/projects/x10/?source=directory.

[28] C. Yilmaz, A. Paradkar, and C. Williams. Time will tell: fault localization using time spectra. In ICSE '08, pages 81–90. ACM, 2008.

[29] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In ICSE '06, pages 272–281. ACM, 2006.