Target-Driven Compositional Concolic Testing with Function Summary Refinement for Effective Bug Detection

Yunho Kim
KAIST
Daejeon, South Korea
yunho.kim03@gmail.com

Shin Hong
Handong Global University
Pohang, South Korea
hongshin@handong.edu

Moonzoo Kim
KAIST
Daejeon, South Korea
moonzoo.kim@gmail.com

ABSTRACT

Concolic testing is popular in unit testing because it can detect bugs quickly in a relatively small search space. But, in system-level testing, it suffers from the symbolic path explosion and often misses bugs. To resolve this problem, we have developed a *focused compositional* concolic testing technique, FOCAL, for effective bug detection. Focusing on a target unit failure v (a crash or an assert violation) detected by concolic unit testing, FOCAL generates a system-level test input that validates v. This test input is obtained by building and solving symbolic path formulas that represent system-level executions raising v. FOCAL builds such formulas by combining function summaries one by one backward from a function that raised v to `main`. If a function summary ϕ_a of function a conflicts with the summaries of the other functions, FOCAL refines ϕ_a to ϕ'_a by applying a refining constraint learned from the conflict. FOCAL showed high system-level bug detection ability by detecting 71 out of the 100 real-world target bugs in the SIR benchmark, while other relevant cutting edge techniques (i.e., AFL-fast, KATCH, Mix-CCBSE) detected at most 40 bugs. Also, FOCAL detected 13 new crash bugs in popular file parsing programs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Automated test generation, target-driven compositional concolic testing, function summary refinement, Craig interpolant, dynamic symbolic execution

ACM Reference Format:
Yunho Kim, Shin Hong, and Moonzoo Kim. 2019. Target-Driven Compositional Concolic Testing with Function Summary Refinement for Effective Bug Detection. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338934

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338934

1 INTRODUCTION

Concolic testing and software model checking have been popular in unit-level [8, 15, 21–23, 26, 27, 35, 39, 40] (i.e., exploring a function f with symbolic unit driver/stubs/environments after separating f from an entire target program P) because they can detect bugs quickly in a relatively small search space [26]. However, concolic unit testing has a weakness, in that most of the detected failures in f are false alarms raised by infeasible unit executions (i.e., unit executions that are infeasible at system-level). This is because concolic unit testing uses approximate symbolic driver/stubs which do not accurately represent the real context(s) of f in P. Note that false alarms are serious obstacles for unit testing [13, 17, 37]. In contrast, system-level concolic testing does not suffer from false alarms because it generates concrete system-level test inputs that execute P from `main`. However, system-level concolic testing often fails to detect bugs due to a huge symbolic path space.

To resolve these weaknesses of unit testing and system testing, we have developed *FOcused CompositionAL concolic testing* (FOCAL). Instead of exploring a huge symbolic search space from scratch to detect failures, FOCAL *identifies target failures quickly* by using concolic unit testing and *to focus on generating system-level test inputs* that validate the failures. Thus, it can detect many bugs in a limited time without false alarms. FOCAL operates as follows:

1. *Identifying target failures v:*
   FOCAL applies concolic unit testing to every function in P and checks whether a failure occurs (i.e., a crash or an assert violation). This unit-level concolic testing can identify many more failures (although many of them are false ones) than system-level concolic testing in a limited testing time.
2. *Generating a system-level test input that validates the identified target failures v:*
   1) For each function a, FOCAL builds a *function summary (FS)* ϕ_a which is a disjunction of the explored symbolic paths (i.e., a under-approximate FS).
   2) FOCAL tries to construct a *validating system-level symbolic path formula* Φ_v whose solution is a system-level test input that validates v. Suppose that a static function call-graph of P has a call-chain from `main` to a_1 where a_1 calls f_v which raised v in concolic unit testing (i.e., $\langle$`main`$, a_n, ..., a_2, a_1\rangle$ such that `main` calls a_n, a_n calls a_{n-1}, and so on).
      - FOCAL obtains Φ_v by combining the summaries of the functions in $\langle$`main`$, a_n, ..., a_2, a_1\rangle$ and the unit executions of f_v that raise v (calling it ψ_v) one by one backward (i.e., combining ψ_v with ϕ_{a_1} first, and then with ϕ_{a_2}, and so on).
      - If an intermediate symbolic path formula (SPF) generated by combining ψ_v with the summaries of functions in the call-chain $\langle a_{k-1}, ..., a_1\rangle$ is satisfiable, but that of a grown

call-chain $\langle \boldsymbol{a_k}, a_{k-1}, ..., a_1 \rangle$ with ψ_v is unsatisfiable, ϕ_{a_k} *conflicts* with the combined summaries of the functions in $\langle a_{k-1}, ..., a_1 \rangle$ and ψ_v. Then, FOCAL *refines* ϕ_{a_k} to ϕ'_{a_k} to resolve the conflict by applying a *refining constraint* which is learned from the conflict by using an SMT solver (i.e., Craig interpolants in Sect. 3.5.4). Note that this target-driven refinement of under-approximate FSes using the Craig interpolants is a new technique and crucial in generating a system-level test input for v (Sect. 3.5.4 and Sect. 6.1).

We performed experiments on the SIR benchmark programs and case studies to detect new crash bugs in real-world file parsing programs. The experiments showed that FOCAL achieved high system-level bug detection ability: it detected 71 out of the 100 real-world target bugs while relevant cutting-edge testing techniques (i.e., AFL-fast, KATCH, Mix-CCBSE) detected only 40, 34, and 25 bugs, respectively (Sect. 5.1). Also, FOCAL successfully detected 13 new crash bugs in popular file parsing programs (Sect. 5.4).

The contributions of this paper are as follows:

- FOCAL is the first technique that detects bugs effectively in a limited testing time by combining the advantages of concolic unit testing (i.e., quick target failure identification) and system-level concolic testing (i.e., validating target failures without false alarms). Without exploring a huge symbolic search space from scratch, it focuses on generating system-level test inputs that validate the target failures identified by concolic unit testing.
- We have developed the following techniques to construct Φ_v effectively and efficiently by composing FSes:
  - Construction of a realistic FS of a function a_i based on a_i's *extended unit*, which provides realistic contexts to a_i (Sect. 3.3). Extended units can reduce false target failures as well as non-validating SPFs (i.e., satisfiable SPFs whose solutions do not validate the target failures) and increase validating SPFs (Sect. 6.2).
  - Target-driven refinement of under-approximate FSes using Craig interpolants to guide concolic testing to construct more validating SPFs (Sect. 3.5.4 and Sect. 6.1).
- We performed systematic empirical evaluations of the bug detection ability of FOCAL and relevant cutting edge testing techniques (i.e., AFL-fast, KATCH, Mix-CCBSE and several variants of FOCAL) on the SIR C programs. In the experiments, FOCAL detected 71 out of the 100 target bugs without any false alarms, while the other techniques detected at most 40 (Sect. 4– 5).
- FOCAL detected 13 new crash bugs in 12 popular file parsing programs. These were reported with the crashing system test inputs generated by FOCAL to the original developers and confirmed by the developers (Sect. 5.4).
- We made 100 real-world bug data for the SIR benchmark programs publicly available (https://sites.google.com/view/focal-fse19). These data were collected and organized after examining the bug reports of the last 12–24 years, so that researchers can use them for various testing research purposes (Sect. 4.2.1).

The paper is organized as follows. Sect. 2 shows an illustrating example. Sect. 3 describes details of FOCAL. Sect. 4 explains the experiment setup used to evaluate FOCAL for comparison with other techniques. Sect. 5 reports the experimental results. Sect. 6 discusses observations from the experiments. Sect. 7 discusses related work. Finally, Sect. 8 concludes the paper with future work.

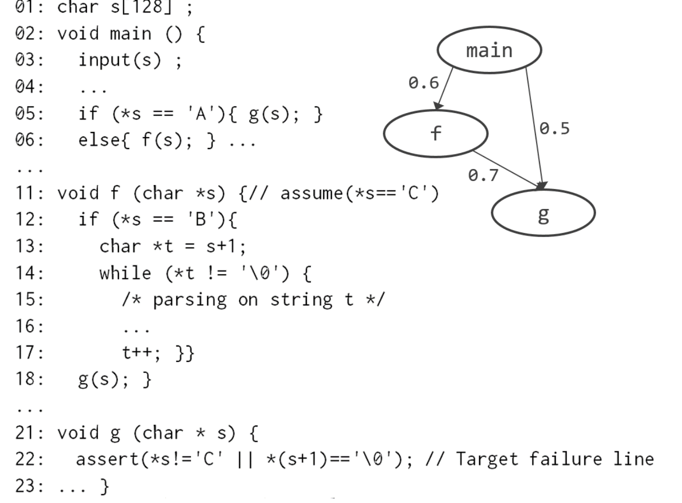

Figure 1: Example target program

2 ILLUSTRATING EXAMPLE

We explain how FOCAL generates a system-level input that fails through an example (three functions `main`, `f`, and `g`, in Fig. 1).

Step 1. Identifying a target failure line v **in** `g`**:**
In Fig. 1, FOCAL identifies Line 22 in `g` as a failure line by concolic unit testing (i.e., v is Line 22 and f_v = `g`). The SPF of the unit execution of f_v that fails at v, ψ_v, is (`*s = 'C'`) ∧ (`*(s+1) ≠ '\0'`).

Step 2. Generating summaries of functions in P **:**
FOCAL generates a FS of every function in P using concolic unit testing. Suppose that, during the concolic unit testing of `f` in a limited testing time, `s` always starts with 'B' (i.e., ∗`s` = `'B'` at Line 12). Then, the FS of `f`, ϕ_f, will be as follows:

$$\begin{aligned} \phi_f &= ((*\mathtt{s} = \text{`B'}) \wedge (*(\mathtt{s}+1) = \text{`\textbackslash 0'})) \\ &\vee ((*\mathtt{s} = \text{`B'}) \wedge (*(\mathtt{s}+1) \neq \text{`\textbackslash 0'}) \wedge \ldots \wedge (*(\mathtt{s}+2) = \text{`\textbackslash 0'})) \\ &\vee \ldots \end{aligned}$$

Step 3. Constructing a system-level symbolic path formula that validates v**:**
To construct a system-level symbolic path formula (SPF) from a target function that raises a failure at v (i.e., `g`) to `main`, FOCAL selects one of the `g`'s callers and combines ψ_v with the FS of the selected caller. Among multiple callers, FOCAL first chooses a caller having the highest function relevance with `g` (the function relevance is given as a label between function nodes in Fig. 1). Among the two callers of `g` (i.e., `f` and `main`), FOCAL first chooses `f` because `g` has a higher function relevance with `f` (i.e., 0.7) than `main` (0.5).

Once `f` is chosen, FOCAL conjoins ψ_v (Step 1) and ϕ_f (Step 2) and finds that $\phi_f \wedge \psi_v$ is unsatisfiable because (`*s = 'B'`) in ϕ_f conflicts with ψ_v = (`*s = 'C'`) ∧ (`*(s+1) ≠ '\0'`).

To refine ϕ_f, first FOCAL obtains a Craig interpolant $\mathcal{I}$ of ψ_v and ϕ_f (i.e.,$\mathcal{I}$:= (`*s = 'C'`)) using Z3. [1] Then, it inserts `assume(`$\mathcal{I}$`)` at the beginning of `f` (i.e., at the end of Line 11) as a refining constraint,

[1] $\mathcal{I}$:= (`*s = 'C'`) is a Craig interpolant of ψ_v and ϕ_f (see Corollary 1) because
- $\models \psi_v \rightarrow \mathcal{I}$ (i.e., $\models$ ((`*s = 'C'`) ∧ (`*(s+1) ≠ '\0'`)) → (`*s = 'C'`), and
- $\mathcal{I} \wedge \phi_f$ is unsatisfiable (i.e., (`*s = 'C'`) ∧ (((`*s = 'B'`) ∧ (`*(s+1) = '\0'`)) ∨ ...))

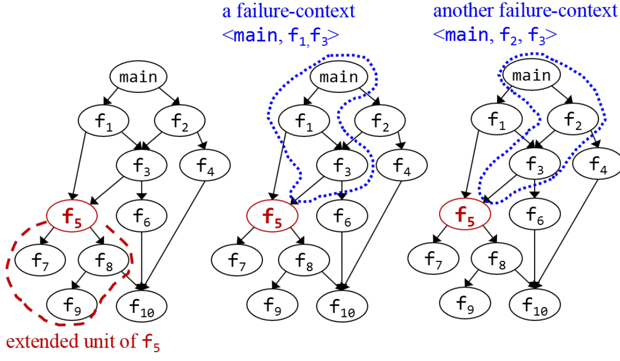

Figure 2: An extended unit of f_5 and two failure contexts $\langle$main, $f_1, f_3\rangle$ and $\langle$main, $f_2, f_3\rangle$ in a static call-graph of P. We assume that f_5 has a target failure line v (i.e., $f_v = f_5$).

where `assume(`*exp*`)` immediately terminates a current execution if *exp* is false. By re-running concolic unit testing of `f` with `assume(*s =='C')` at Line 11, FOCAL obtains a refined FS ϕ'_f := (`*s = 'C'`).

Now, $\phi'_f \wedge \psi_v$ = (`*s = 'C'`) $\wedge$ ((`*s = 'C'`) $\wedge$ (`*(s+1)` $\neq$ `'\0'`)) becomes satisfiable. Then, FOCAL continues to construct a system-level SPF for the failure by conjoining $\phi_{\texttt{main}}$ with $\phi'_f \wedge \psi_v$. Finally, if the final SPF Φ_v (= $\phi_{\texttt{main}} \wedge \phi'_f \wedge \psi_v$) is satisfiable, FOCAL generates a system-level input (i.e., a solution of Φ_v) that crashes the program at Line 22.

If Φ_v is not satisfiable, FOCAL tries to refine $\phi_{\texttt{main}}$ into $\phi'_{\texttt{main}}$ then it checks if Φ'_v (= $\phi'_{\texttt{main}} \wedge \phi'_f \wedge \psi_v$) is satisfiable. If Φ'_v is not satisfiable, FOCAL tries to build a different validating SPF by backtracking to combine ψ_v and $\phi_{\texttt{main}}$ (instead of ϕ_f) and checks if $\phi_{\texttt{main}} \wedge \psi_v$ is unsatisfiable (it refines $\phi_{\texttt{main}}$ into $\phi''_{\texttt{main}}$ if necessary). If $\phi_{\texttt{main}} \wedge \psi_v$ (and $\phi''_{\texttt{main}} \wedge \psi_v$) is still unsatisfiable, FOCAL tried all possible call-chains from `main` to `g`, but failed to generate a test input to validate the target failure v at Line 22.

3 FOCUSED COMPOSITIONAL CONCOLIC TESTING TECHNIQUE (FOCAL)

This section explains how FOCAL operates using a target program P in Fig. 2 as an example. Fig. 2 shows a static call-graph of P which consists of 11 functions (i.e., `main`, f_1, ..., and f_{10}) including the program entry function `main`.

3.1 Overview

Fig. 3 shows the overall process of FOCAL. FOCAL takes as inputs a target program P and a set of system-level seed test inputs T and generates system-level test inputs that make P fail. FOCAL operates in the following four phases (see Fig. 3):

1. Measuring function relevance (Sect. 3.2):
   First, FOCAL generates system-level test inputs by fuzzing the given system-level seed test inputs. Then, from the function call profile obtained by executing the fuzzed system tests, it measures relevance between every pair of the functions in P. This information is used for target failure line identification, FS generation, and system-level SPF construction.
2. Identification of a *target failure line* v (Sect. 3.3):
   To identify a *target failure line* v (i.e., a line of program code where a crash or an assert violation occurs), FOCAL applies concolic unit testing to every function in P. We call a function that has v as f_v.
   For example, FOCAL applies concolic unit testing to each of `main`, f_1, ..., and f_{10} in Fig. 2 separately. Suppose that f_5 is written in Lines 30–50 and f_5 crashes at Line 40 during concolic unit testing of f_5. Then, we set Line 40 as a target failure line v and $f_v = f_5$.
3. Construction of *function summaries* (Sect. 3.4):
   For each function a in P, FOCAL builds a FS ϕ_a which is a disjunction of SPFs explored by concolic unit testing (i.e., an under-approximate FS).
4. Construction of *system-level SPFs* to validate a failure at v (Sect. 3.5):
   To validate a failure at v in system-level (i.e., generating a test input that makes P fail at v), FOCAL builds SPFs by combining unit failure executions ψ_v (i.e., a set of unit executions of f_v that fail at v) and the summaries of the functions in v's *failure-context* (e.g., $\langle a_k, ..., a_1\rangle$ which is a call-chain to a_1 that calls f_v).
   For example of Fig. 2, suppose that concolic unit testing of f_5 crashes at Line 40 in f_5 (i.e., $f_v = f_5$). Then, FOCAL builds SPFs by combining ψ_v and the summaries of the functions in the failure-context of v (e.g., $\langle f_1\rangle$, $\langle$main, $f_1\rangle$, $\langle f_3\rangle$, $\langle f_1, f_3\rangle$, $\langle$main, $f_1, f_3\rangle$, $\langle f_2, f_3\rangle$, and $\langle$main, $f_2, f_3\rangle$) until it constructs a validating SPF whose solution makes P fail at v.

3.2 Function Relevance Metric

FOCAL computes function relevance metric using the conditional probability based on the function call profiles observed from system test executions. To obtain accurate function relevance, FOCAL generates a large number of system test inputs by fuzzing a given set of system-level seed test inputs T. FOCAL considers that f and g are highly relevant if it frequently observes that f calls g (immediately or transitively) (denoted by $f \rightarrow g$) or g calls f in system test executions. Intuitively speaking, if caller-callee functions execute together frequently, they closely interact with each other, which means that they are highly relevant to each other.

We measure the relevance between f and g (denoted by $r(f, g)$) as $\frac{(x+y)}{2}$ such that

- x is $p((f \rightarrow g \text{ or } g \rightarrow f)|f)$ which is calculated by $\frac{w}{z}$ where
  - w is the number of the system test executions where $f \rightarrow g$ or $g \rightarrow f$ occurs
  - z is the number of the system test executions where f occurs
- y is $p((f \rightarrow g \text{ or } g \rightarrow f)|g)$

Ex. Suppose that we have the following test executions in Fig. 2:

- $t1$ = {main $\rightarrow f_1$, $f_1 \rightarrow f_5$, $f_5 \rightarrow f_7$}
- $t2$ = {main $\rightarrow f_1$, $f_1 \rightarrow f_5$, $f_5 \rightarrow f_8$, $f_8 \rightarrow f_9$}
- $t3$ = {main$\rightarrow f_1$, $f_1 \rightarrow f_5$,$f_5 \rightarrow f_7$, $f_5 \rightarrow f_8$,$f_8 \rightarrow f_9$, $f_8 \rightarrow f_{10}$}
- $t4$ = {main $\rightarrow f_1$, $f_1 \rightarrow f_3$, $f_3 \rightarrow f_6$, $f_6 \rightarrow f_{10}$}

In this example, $r(f_5, f_7)$=0.83 (=($\frac{2}{3}$+1)/2 because $p(f_5 \rightarrow f_7|f_5) = \frac{2}{3}$ and $p(f_5 \rightarrow f_7|f_7) = 1$) and $r(f_5, f_{10})$= 0.42 (= ($\frac{1}{3}$ + $\frac{1}{2}$)/2 because $p(f_5 \rightarrow f_{10}|f_5) = \frac{1}{3}$ and $p(f_5 \rightarrow f_{10}|f_{10}) = \frac{1}{2}$).

3.3 Identification of a Target Failure Line v

FOCAL applies concolic unit testing to each function a in P and identifies a target line v in a if a fails. To reduce false target lines

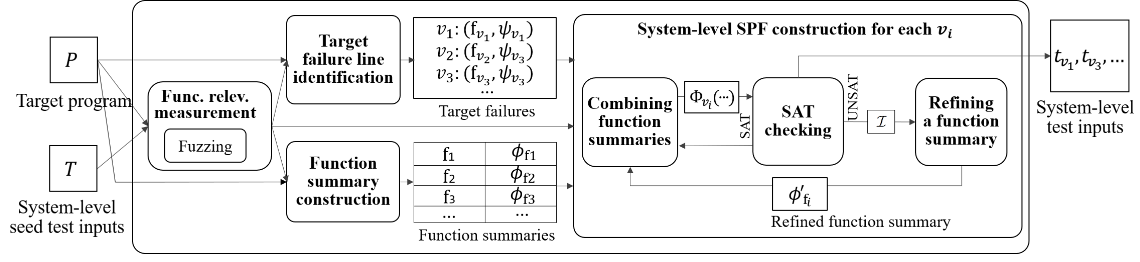

Figure 3: Focused Compositional Concolic Testing (FOCAL)

caused by infeasible unit executions, FOCAL applies concolic unit testing to an *extended unit* of a (denoted by $E(a)$) [23]. $E(a)$ consists of a, a's *closely relevant* callee functions $b_1 ... b_n$ (i.e., functions reachable from a in a static call-graph), and symbolic stubs to replace callee functions of a that are not closely relevant to a (symbolic stubs return unconstrained symbolic-values).

Note that concolic unit testing $E(a)$ (instead of a alone) can reduce false alarms by removing infeasible unit executions of a because a's closely related functions can provide realistic environment to a. The relevance between functions a and b (denoted by $r(a,b)$) is measured based on how frequently a calls b or vice versa, among system test executions (Sect. 3.2). Note that $E(a)$ should *only* contain functions $b_1 ... b_n$ which are closely relevant to a (i.e., $r(a,b)$ is among the top 30% of the relevancies between all function pairs in P) since including more functions will enlarge symbolic path space and degrade unit testing effectiveness and efficiency.

For example of Fig. 2, suppose that $E(f_5)$ is $\{f_5, f_7, f_8, f_9\}$ because $r(f_5, f_7)$ is high (i.e., among the top 30% of the relevancies between all function pairs in P). Similarly, suppose that $r(f_5, f_8)$ and $r(f_5, f_9)$ are also high, but not $r(f_5, f_{10})$. When FOCAL applies concolic unit testing to f_5, it explores $E(f_5)$ consisting of $\{f_5, f_7, f_8, f_9\}$.

3.4 Function Summary Construction

FOCAL builds a FS of every function a of P as a disjunction of all explored SPFs σ_is on $E(a)$ (i.e., $\phi_a \overset{def}{=} \bigvee \sigma_i$) during the target failure line identification process (Sect. 3.3). This approach of building FSes is applicable for complex real-world programs with nested loops, external binary libraries, complex pointer arithmetic, etc. The syntax and semantics of FS follow QF_BV in SMTLIB2.

To focus on exploring diverse behaviors of a in a given time budget, FOCAL applies concolic unit testing to $E(a)$ using a weighted random negation search strategy. This search strategy randomly negates a branch in a current symbolic path while giving four times higher chance to the branches in a than the branches in the other functions in $E(a)$.

3.5 Construction of System-level SPFs to Validate a Failure at v

3.5.1 Preliminaries.

- f_v is the function which has a target failure line v.
- C_v is a set of functions that directly call f_v. For example of Fig. 2, $f_v = f_5$ and $C_v = \{f_1, f_3\}$.
- ψ_v is a set of failure executions in f_v(i.e., a disjunction of the SPFs of $E(f_v)$ (generated by concolic unit testing) that fail at v).
- ϕ_a denotes a FS of a function a. ϕ_a is a disjunction of the SPFs of $E(a)$ (generated by concolic unit testing) (i.e., $\phi_a = \bigvee \sigma_i$ where σ_i is a SPF of $E(a)$).
- A *failure-context* $S_v = \langle a_k, ..., a_2, a_1 \rangle$ of a target failure line v is a call-chain/path in a static call-graph of P such that a_k calls a_{k-1}, a_{k-1} calls a_{k-2} and so on and $a_1 \in C_v$. For example, the failure-contexts of v in Fig. 2 are $\langle f_1 \rangle$, $\langle \mathtt{main}, f_1 \rangle$, $\langle f_3 \rangle$, $\langle f_1, f_3 \rangle$, $\langle \mathtt{main}, f_1, f_3 \rangle$, $\langle f_2, f_3 \rangle$, and $\langle \mathtt{main}, f_2, f_3 \rangle$.
- $Slice(\phi_{a_{i+1}}, a_i)$ is a sliced formula of ϕ_{a_i+1} with regard to the invocation of a function a_i (i.e., for $\phi_{a_{i+1}} = \bigvee \sigma_j$, $Slice(\phi_{a_{i+1}}, a_i) = \bigvee \sigma'_j$ where σ'_j is a prefix of σ_j only up to an invocation of a_i).
- $\Phi_v(\langle a \rangle)$ for $a \in C_v$ denotes a combined SPF of ψ_v and the FS of a that directly calls f_v (i.e., $\Phi_v(\langle a \rangle) = Slice(\phi_a, f_v) \wedge \psi_v$).
- For a failure-context $S_v^k = \langle a_k, ..., a_1 \rangle$, $\Phi_v(S_v^k)$ denotes a combined SPF of ψ_v and ψ_v's symbolic calling context formula which is the combined sliced summaries of the functions in S_v in a backward order (i.e., $\Phi_v(S_v^k) = Slice(\phi_{a_k}, a_{k-1}) \wedge \Phi_v(\langle a_{k-1}, ..., a_1 \rangle)$). Combined FSes capture effects on visible variables, parameters, return-values in SSA form (i.e., all variables in FSes are expressed as expressions over the symbolic input-variables).
  For example of Fig. 2,

$$\begin{aligned}\Phi_v(\langle \mathtt{main}, f_1, f_3 \rangle) &= Slice(\phi_{\mathtt{main}}, f_1) \wedge \Phi_v(\langle f_1, f_3 \rangle)\\ &= Slice(\phi_{\mathtt{main}}, f_1) \wedge Slice(\phi_{f_1}, f_3) \wedge \Phi_v(\langle f_3 \rangle)\\ &= Slice(\phi_{\mathtt{main}}, f_1) \wedge Slice(\phi_{f_1}, f_3) \wedge Slice(\phi_{f_3}, f_5) \wedge \psi_v\end{aligned}$$

3.5.2 Strategies for Function Summary Composition. To generate SPFs that validate a failure at v quickly, FOCAL uses function relevance metric to select a FS to combine as follows (i.e., giving a high priority to a function which has high relevance with a most recently combined function). Suppose that FOCAL has built $\Phi_v(S_v^{k-1})$ where the failure-context S_v^{k-1} is $\langle a_{k-1}, ..., a_1 \rangle$ and a_{k-1} is called by b_1,...,b_m. FOCAL selects b_i whose relevance with a_{k-1} is the highest and combines ϕ_{b_i} with $\Phi_v(S_v^{k-1})$. If FOCAL fails to generate Φ_v after selecting ϕ_{b_i} to combine with $\Phi_v(S_v^{k-1})$, it backtracks to select and combine $b_{i'}$ which has the second highest relevance with a_{k-1} and so on.

This function relevance-based FS composition can be effective because, if b is more relevant with a_{k-1}than b', it will be easier to refine ϕ_b to be compatible with a_{k-1} than $\phi_{b'}$ because b and a_{k-1} share more common contexts than b' and a_{k-1} (i.e., ϕ_b may need less refinement steps to become compatible with a_{k-1} than $\phi_{b'}$).

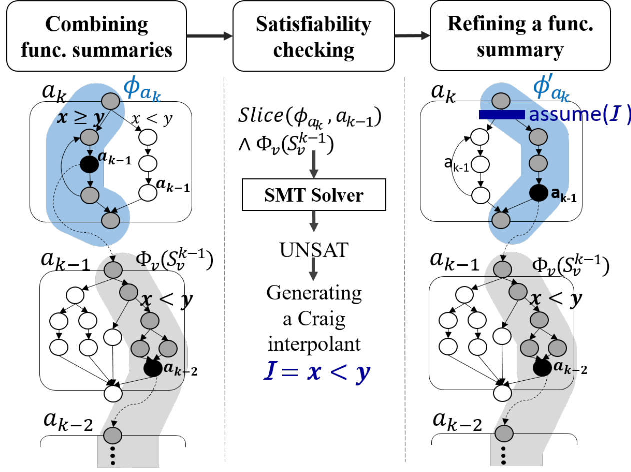

Figure 4: FSR using the Craig interpolants

3.5.3 Generation of Symbolic Path Formulas by Combining Function Summaries. To generate a SPF Φ_v to validate a failure at v, FOCAL combines the summaries of the functions in a failure-context $\langle \texttt{main}, ..., a_1\rangle$ of v and ψ_v in a backward order. If the combined SPF is satisfiable, FOCAL uses a solution of the formula obtained by an SMT solver as a system-level test input to validate a failure at v.

For Fig. 2 where $f_v = f_5$, FOCAL generates SPFs as follows:

1. Suppose that $r(f_3, f_5) > r(f_1, f_5)$. FOCAL generates $\Phi_v(\langle f_3\rangle) = Slice(\phi_{f_3}, f_5) \wedge \psi_v$. If $\Phi_v(\langle f_3\rangle)$ is satisfiable and $r(f_1, f_3) > r(f_2, f_3)$, FOCAL increases a failure-context of v to $\langle f_1, f_3\rangle$
2. It generates $\Phi_v(\langle f_1, f_3\rangle) = Slice(\phi_{f_1}, f_3) \wedge \Phi_v(\langle f_3\rangle)$ and checks if $\Phi_v(\langle f_1, f_3\rangle)$ is satisfiable. If yes, FOCAL increase a failure-context of v to $\langle \texttt{main}, f_1, f_3\rangle$.
3. Finally, if $\Phi_v(\langle \texttt{main}, f_1, f_3\rangle) = Slice(\phi_{\texttt{main}}, f_1) \wedge \Phi_v(\langle f_1, f_3\rangle)$ is satisfiable, FOCAL obtains a solution of $\Phi_v(\langle \texttt{main}, f_1, f_3\rangle)$ by using a SMT solver and uses the solution as a system-level test input to validate a failure at v.

Meanwhile, a combined SPF may be *unsatisfiable* if a FS conflicts with the other FSes. For example, suppose that a_k calls a_{k-1} in a failure-context of v (i.e., $S_v^{k-1} = \langle a_{k-1}, ..., a_1\rangle$) and ϕ_{a_k} does not contain a symbolic path that provides a context necessary for $\Phi_v(S_v^{k-1})$ to invoke a failure at v. Then, the combined formula $Slice(\phi_{a_k}, a_{k-1}) \wedge \Phi_v(S_v^{k-1})$ will be unsatisfiable. In such cases, FOCAL refines ϕ_{a_k} as shown in Sect. 3.5.4.

3.5.4 Function Summary Refinement (FSR). Fig. 4 shows how FOCAL refines a FS. Suppose that FOCAL combined ϕ_{a_k} and $\Phi_v(S_v^{k-1})$ where a failure-context S_v^{k-1} is $\langle a_{k-1}, ..., a_1\rangle$, a_k calls a_{k-1} and $\Phi_v(S_v^{k-1})$ is satisfiable. If the combined formula is unsatisfiable due to the conflict between ϕ_{a_k} and $\Phi_v(S_v^{k-1})$, to continue construction of SPFs to validate a failure at v, FOCAL refines ϕ_{a_k} into ϕ'_{a_k}. It builds ϕ'_{a_k} using concolic testing [2] on $E(a_k)$ with a Craig interpolant of $\Phi_v(S_v^{k-1})$ and $Slice(\phi_{a_k}, a_{k-1})$ as a refining constraint. Craig interpolation theorem is given as follows:

[2] To build a refined FS quickly, FOCAL extends CFG search heuristic [7] to guide the search to reach the lines where a_k calls a_{k-1} quickly.

Theorem 1 (Craig, 1957 [9]). *Suppose $A \rightarrow C$ is a valid implication in first-order logic (i.e., $\models A \rightarrow C$). Then, there is a Craig interpolant $\mathcal{I}$ such that $\models A \rightarrow \mathcal{I}$ and $\models \mathcal{I} \rightarrow C$.*

Corollary 1. *Suppose that $A \wedge B$ is unsatisfiable in first-order logic (i.e., $\models A \rightarrow \neg B$). Then, by Thm. 1, there is a Craig interpolant $\mathcal{I}$ such that $\models A \rightarrow \mathcal{I}$ and $\mathcal{I} \wedge B$ is unsatisfiable.* [3]

Suppose that A is $\Phi_v(S_v^{k-1})$, B is $Slice(\phi_{a_k}, a_{k-1})$, and $\Phi_v(S_v^{k-1}) \wedge Slice(\phi_{a_k}, a_{k-1})$ is unsatisfiable. Then, by Corollary 1, there exists a Craig interpolant $\mathcal{I}$ of $\Phi_v(S_v^{k-1})$ and $Slice(\phi_{a_k}, a_{k-1})$ such that $\mathcal{I} \wedge Slice(\phi_{a_k}, a_{k-1})$ is unsatisfiable. Note that $Slice(\phi_{a_k}, a_{k-1})$ represents the already explored paths in a_k. Thus, Craig interpolant $\mathcal{I}$ can work as a guide in concolic unit testing of a_k to avoid revisiting already explored paths (i.e., $\mathcal{I} \rightarrow \neg Slice(\phi_{a_k}, a_{k-1})$). And at the same time, $\mathcal{I}$ can lead the concolic unit testing to explore paths compatible with $\Phi_v(S_v^{k-1})$ (i.e., $\Phi_v(S_v^{k-1}) \rightarrow \mathcal{I}$).

Now we propose the following heuristic to build ϕ'_{a_k} such that $\Phi_v(\langle a_k, ..., a_1\rangle)$ is satisfiable.

- When FOCAL generates ϕ'_{a_k} using concolic testing, FOCAL enforces a Craig interpolant $\mathcal{I}$ of $\Phi_v(S_v^{k-1})$ and $Slice(\phi_{a_k}, a_{k-1})$ as a FS refining constraint so that ϕ'_{a_k} can be different from ϕ_{a_k} (and, thus, ϕ'_{a_k} may not conflict with $\Phi_v(S_v^{k-1})$).

This strategy constructs a new FS ϕ'_{a_k} that can be compatible with $\Phi_v(S_v^{k-1})$. This is because $\mathcal{I}$ guides concolic testing to make ϕ'_{a_k} contain symbolic paths *different from* the ones in ϕ_{a_k} by pruning $Slice(\phi_{a_k}, a_{k-1})$ (because $\mathcal{I} \wedge Slice(\phi_{a_k}, a_{k-1})$ is unsatisfiable).

FOCAL implements this strategy by inserting `assume`($\mathcal{I}$) at the beginning of the body of a_k, which guides concolic execution to explore paths that satisfy $\mathcal{I}$ by terminating an execution of a_k immediately if $\mathcal{I}$ is violated.

Suppose that ϕ'_{a_k} does not resolve the conflict in the first function summary refinement step. We call the Craig interpolant used in this first function summary refinement step as $\mathcal{I}^1$ and the first refined FS as $\phi^1_{a_k}$ (=ϕ'_{a_k}). Then FOCAL obtains the second interpolant $\mathcal{I}^2 = \mathcal{I}(\Phi_v(S_v^{k-1}), Slice(\phi^1_{a_k}, a_{k-1}))$. Then, it builds a new refined FS $\phi^2_{a_k}$ using $\mathcal{I}^2 \wedge \mathcal{I}^1$ as a new refining constraint and checks whether $\phi^2_{a_k}$ resolves the conflict. If not, this step repeats until a newly refined FS does not increase branch coverage of a_k three times in a row (i.e., until this step does not explore new search space of a_k much) or the conflict is resolved.

3.5.5 Example of Constructing Symbolic Path Formulas to Validate a failure at v in Fig. 2. Suppose that $f_v = f_5$ and $r(f_5, f_3) = 0.9$, $r(f_5, f_1) = 0.8$, $r(f_3, f_1) = 0.7$, and $r(f_3, f_2) = 0.6$.

First, FOCAL selects f_3 to build $\Phi_v(\langle f_3\rangle)$ by combining ϕ_{f_3} and ψ_v because $r(f_5, f_3) > r(f_5, f_1)$. Suppose that $\Phi_v(\langle f_3\rangle)$ is satisfiable. Then, FOCAL continues to select f_1 (because $r(f_3, f_1) > r(f_3, f_2)$) to combine ϕ_{f_1} and $\Phi_v(\langle f_3\rangle)$ and obtains $\Phi_v(\langle f_1, f_3\rangle)$. Suppose that $\Phi_v(\langle f_1, f_3\rangle)$ is unsatisfiable. Then, FOCAL refines ϕ_{f_1} into ϕ'_{f_1} by using a Craig interpolant $\mathcal{I}(\Phi_v(\langle f_3\rangle), Slice(\phi_{f_1}, f_3))$. Suppose that $\Phi'_v(\langle f_1, f_3\rangle) = Slice(\phi'_{f_1}, f_3) \wedge \Phi_v(\langle f_3\rangle)$ is satisfiable. Then, FOCAL

[3] Corollary 1 is just another form of Theorem 1. $\models A \rightarrow C$ in Thm 1 is equivalent to that $A \wedge B$ is unsatisfiable in Cor 1, because if we replace C in Thm 1 with $\neg B$ in Cor 1, then $\models A \rightarrow \neg B \equiv\, \models \neg A \vee \neg B \equiv\, \models \neg(A \wedge B)$. Also, $\models \mathcal{I} \rightarrow C$ in Thm 1 is equivalent to $\models \mathcal{I} \rightarrow \neg B$ in Cor 1 which indicates that $\mathcal{I} \wedge B$ is unsatisfiable because $\models \mathcal{I} \rightarrow \neg B \equiv\, \models \neg \mathcal{I} \vee \neg B \equiv\, \models \neg(\mathcal{I} \wedge B)$.

Table 1: Programs of the known crash bug benchmark

Target programs	Lines	# of func.	# of seed sys. TCs	Branch cov.(%)	Func cov. (%)	# of target bugs
Bash-2.0	32714	1214	1100	46.2	89.0	39
Flex-2.4.3	7471	147	567	45.7	93.9	5
Grep-2.0	5956	132	809	50.3	94.7	6
Gzip-1.0.7	3054	82	214	55.8	87.8	5
Make-3.75	28715	555	1043	64.5	87.9	10
Sed-1.17	4085	73	360	47.3	87.7	3
Vim-5.0	66209	1749	975	35.8	91.0	32
Sum	148204	3952	5068	N/A	N/A	100
Average	21172.0	564.6	724.0	49.4	90.3	14.3

finally combines $\phi_{\texttt{main}}$ and $\Phi'_v(\langle f_1, f_3\rangle)$ to build $\Phi_v(\langle \texttt{main}, f_1, f_3\rangle)$ (see a dotted circle on a failure-context $\langle \texttt{main}, f_1, f_3\rangle$ in the middle of Fig.2). However, suppose that $\Phi_v(\langle \texttt{main}, f_1, f_3\rangle)$ is unsatisfiable and FOCAL fails to refine $\phi_{\texttt{main}}$ to be compatible with $\Phi'_v(\langle f_1, f_3\rangle)$.

Then, FOCAL backtracks to f_3 and combines f_2 (instead of f_1) with $\Phi_v(\langle f_3\rangle)$ to build $\Phi_v(\langle f_2, f_3\rangle)$. Suppose that $\Phi_v(\langle f_2, f_3\rangle)$ is satisfiable. Then, FOCAL continues to build $\Phi_v(\langle \texttt{main}, f_2, f_3\rangle)$ (see a dotted circle on a failure-context $\langle \texttt{main}, f_2, f_3\rangle$ in the right part of Fig.2). If $\Phi_v(\langle \texttt{main}, f_2, f_3\rangle)$ is satisfiable, FOCAL generates a solution to the formula and uses the solution as a system-level test input to validate a failure at v.

4 EXPERIMENT SETUP

4.1 Research Questions

RQ1. Bug detection ability: How many target bugs does FOCAL detect, compared to fuzzing (AFL-fast [5]) and the guided concolic testing techniques (KATCH [31] and Mix-CCBSE [30])?

RQ2. Effect of the Craig interpolants in FSR: How much does the Craig interpolants in function summary refinement (FSR) affect the number of the detected bugs and the execution time to build SPFs?

RQ3. Effect of the extended units for bug detection and execution time : How much do the extended units affect FOCAL's number of the bugs detected and execution time, compared to FOCAL without the extended units (FOCAL^{-E}) and FOCAL with randomly built extended units (FOCAL^{R})?

RQ4. New crash bug detection: How many new bugs does FOCAL detect, compared to AFL-fast, KATCH, and Mix-CCBSE?

4.2 Target Bugs

To measure bug detection ability of the techniques, we target crash bugs (although FOCAL can detect non-crash bugs if test oracles are provided as assertions) because they (e.g., null-pointer dereference, divide-by-zero, buffer overflow) cause serious reliability and security problems and can be detected with automatically generated assertions.

FOCAL automatically inserts assertions (e.g., `assert(ptr!=NULL)`) to detect crash bugs in the target programs.

We use two sets of target programs: *known crash bug benchmark* for RQ1-3 and *new crash bug benchmark* for RQ4.

4.2.1 Known Crash Bug Benchmark. We collected 100 real-world crash bugs of the seven SIR C programs (shown in Table 1) that are larger than 1 KLoC and were fixed by the original developers from Dec 1996 to July 2018. Each program is the same version of the programs in SIR [12] because they are widely used for the software testing research.

From the revision histories, we collected bugs such that (1) the bug report shows that the bug crashes the program, (2) the original developers confirmed the bug report and released a patch to fix the bug, and (3) the bug exists at the version chosen for the benchmark (i.e., the same version in SIR). We collected total 19,108 bug-fix commits of the target programs. Then, we extracted 587 crash bug-fix commits by searching keywords like "overflow", "segfault", etc. We manually analyzed the changed code and commit logs of the 587 crash bug-fix commits and identified 100 crash bugs.

We consider a program line l_b as a faulty line of a bug b if l_b is included in the patch (i.e., the bug-fix commits for b). We did not use any artificially inserted bugs in SIR.

4.2.2 New Crash Bug Benchmark. To evaluate the effectiveness of FOCAL for discovering new crash bugs, we target the popular C programs that parse regular expression, XML and JSON [6]. We choose the latest versions of the target text parsing C programs as of July 2018. The programs consist of 7243.8 LoC and 272.3 functions on average (details of the new crash bug target programs are available at https://sites.google.com/view/focal-fse19). The text parsing libraries are widely used in various software including server applications and smartphone apps and the crash bugs in these libraries can cause severe reliability and security problems.

4.3 FOCAL Setup

4.3.1 Fuzzing . To compute function relevance from diverse system behaviors, FOCAL applied the AFL-fast fuzzer [5] to generate various system test inputs. Using all system tests provided in a target program as seed test inputs, FOCAL ran AFL-fast for 1 hour (no target bug detected).

For known crash bug target programs, it generated 24,300 system test inputs that executed previously unexplored execution paths on average per program (achieving 79.3% branch coverage). [4] For new crash bug target programs, it generated 33,300 system test inputs that executed previously unexplored execution paths on average per program (achieving 90.3% branch coverage).

4.3.2 Construction of Extended Units. For each function a, FOCAL constructs an extended unit of a (i.e., $E(a)$) based on the function relevance between a and a's (immediate or transitive) callee function b (i.e., $r(a, b)$). If $r(a, b)$ is in the top 30% of the relevancies of all pairs of functions in P (i.e., b is closely relevant to a), b is included in $E(a)$; if not, b is not included in $E(a)$ and replaced by a symbolic stub function.

4.3.3 Timeout of Concolic Unit Testing. For concolic unit testing for target failure line identification (Sect. 3.3), FS construction (Sect. 3.4),

[4] The quality of the seed test inputs does not affect bug detection ability of FOCAL much because it uses diverse test inputs generated by fuzzing the seed test inputs. For example, when we randomly selected and used only 10% of the seed test inputs (achieving branch coverage 28.3% on average) per program, FOCAL still detected 69 out of 100 target bugs.

and FSR (Sect. 3.5.4), we set ten minutes timeout for each function in P.

4.3.4 Implementation.

We have implemented FOCAL and its variants in 7,800 lines of C++ code using Clang/LLVM-4.0 [29]. FOCAL uses AFL-fast [5] for fuzzing, CROWN [25] for concolic testing and Z3 [10] for solving SMT constraints and computing Craig interpolants.

4.4 Automated Test Generation Techniques to Compare

We have evaluated FOCAL and the following testing techniques:

- *Fuzzing technique (AFL-fast [5])*: AFL-fast guides the search-based fuzzing to cover rarely explored code locations. We used all system test inputs generated to compute function relevance as seed test inputs for AFL-fast. We set the timeout of AFL-fast as the same amount of the total execution time of FOCAL.
- *Directed concolic testing techniques (KATCH and Mix-CCBSE)*: KATCH [31] takes a program, a patch, and a set of regression tests to generate test inputs to cover the code locations changed by the patch. To guide KATCH to execute the target failure line v identified by concolic unit testing of FOCAL, we make a patch that adds a crash assertion at v to a target program. Mix-CCBSE [30] takes a program and a target line as inputs and performs concolic testing to cover the target code lines. We give each of the target failure lines v identified by concolic unit testing of FOCAL to Mix-CCBSE as the target line. We implemented our own prototype of Mix-CCBSE on KLEE 1.4 (in 600 lines of C++ code) since we could not use the Mix-CCBSE implementation due to technical problems (the implementation has not been maintained since 2013).
  We set the timeout of KATCH and Mix-CCBSE for each target failure line v as the same amount of the execution time spent for the most time-consuming target failure line of P by FOCAL. For example, if FOCAL spends one hour to validate the most time-consuming target failure line in P, we give one hour to KATCH and Mix-CCBSE for each target failure line in P.
- *FOCAL$^{-\mathcal{I}}$*: it is a variant of FOCAL that performs FSR *without* the Craig interpolants. For fair comparison with FOCAL, FOCAL$^{-\mathcal{I}}$ builds a refined FS in 90 minutes, which is more than the largest amount of time (87 minutes) spent by repeated FSRs using the Craig interpolants (Sect. 3.5.4).
- *FOCAL^{-E}*: it is a variant of FOCAL that does not use extended units (i.e., concolic unit testing performs on a single function a with symbolic stubs that replace all callee functions of a).
- *FOCAL^{R}*: it is a variant of FOCAL that uses randomly constructed extended units (i.e., $E(a)$ contains a and *randomly selected* callee functions of a (with a probability 0.5), and symbolic stubs that replace the other callee functions of a). For example of Fig. 2, suppose that FOCAL^R randomly adds f_8 to $E(f_5)$ but not f_7. Then, it continues to randomly add f_{10} (a callee of f_8) to $E(f_5)$ but not f_9. As a result, FOCAL^R constructs $E(f_5)$ as $\{f_5, f_8, f_{10}\}$.

4.5 Measurement

To reduce the random variance on the experiment, we repeated the experiments ten times and report the average numbers.

4.5.1 Bug Detection.

For a known crash bug b, we report that b is *detected* if a technique generates a system-level test input that makes P reach l_b (one of the faulty code lines of b) and then crash at a target failure line. If one system execution has covered the faulty lines of multiple target bugs, we manually analyzed the system execution to identify which bug causes the failure at a target failure line.

For a new crash bug, we report the number of the target failure lines where crashes are validated by the generated system test inputs as the number of detected bugs. This is because we do not know which bug covers which failure line(s).

4.5.2 Execution Time.

We report the execution time of a technique on a single machine for a fair comparison of FOCAL with other testing techniques. The execution time of FOCAL and its variants consists of:

- Fuzzing and function relevance measurement (FZ): one hour spent to fuzz the seed test inputs (and negligible amount of time to calculate the function relevance using the fuzzed test inputs)
- Target failure line identification (FLI): time spent by concolic unit testing each $E(a)$ to identify target failure lines
- Satisfiability check (SC): time spent for checking satisfiability of constructed SPFs (Sect. 3.5.3)
- Craig interpolant calculation (CC): time spent for computing the Craig interpolants for FSR
- Function summary refinement (FSR): time spent for running concolic unit testing to obtain a refined FS

In RQ1 and RQ4, we report the sum of FZ, FLI, SC, CC, and FSR time as the execution time of FOCAL to compare with AFL-fast, KATCH, and Mix-CCBSE. In RQ2 to RQ3, we report the sum of SC, CC, and FSR time because FOCAL and its variants share the same target failure lines and have the same amount of FZ and FLI time.

4.6 Testbed Setting

Since the experiment scale is large, the experiments were performed on 30 machines equipped with Intel quad-core i5 4670K (3.4 Ghz) and 8GB RAM, running Ubuntu 16.04 64 bit version. Each machine runs four instances of testing processes.

4.7 Threats to Validity

A threat to external validity is the representativeness of our target programs. We expect that this threat is limited since the target programs are widely used real-world ones and tested by many other researchers. Also, the set of target bugs might not be complete because we might fail to extract one from the bug reports or a target program has an unknown (i.e., not reported) bug. We expect that this threat is also limited because we did our best to thoroughly review the bug reports and the target programs are actively maintained. A threat to internal validity is possible bugs in the implementations of FOCAL and the other concolic testing techniques we studied. We extensively tested our implementations to address this threat.

5 EXPERIMENT RESULTS

This section presents experiment results to answer the research questions. All detailed data are available at https://sites.google.com/view/focal-icse19.

Table 2: # of the target bugs detected by and the execution time (hours) on a single machine of AFL-fast, KATCH, Mix-CCBSE, and FOCAL

Targets	AFL-fast		KATCH		Mix-CCBSE		FOCAL	
	#det. bugs	Time(h)	#det. bugs	Time(h)	#det. bugs	Time(h)	#det. bugs	Time(h)
Bash	11	399.1	10	522.7	8	669.2	25	399.1
Flex	2	125.2	2	164.0	1	176.0	4	125.2
Grep	4	250.8	3	357.8	2	400.1	5	250.8
Gzip	4	112.5	3	181.1	2	227.2	4	112.5
Make	6	294.7	4	347.4	3	479.4	9	294.7
Sed	3	134.6	2	205.4	2	225.3	3	134.6
Vim	10	521.5	10	648.9	7	746.9	21	521.5
Sum	40	1838.3	34	2427.3	25	2924.0	71	1838.3
Avg.	5.7	262.6	4.9	346.8	3.6	417.7	10.1	262.6

5.1 RQ1: Bug Detection Ability

FOCAL showed high bug detection ability. Table 2 shows the numbers of the target bugs detected by and the execution time of AFL-fast, KATCH, Mix-CCBSE, and FOCAL. It shows the total execution time (in hours) spent on a single machine (for fair comparison between the techniques). The wall-clock execution time is roughly 1/100 of the reported time (e.g., FOCAL spent 18 hours to perform all experiment) because the experiment was run on 30 quad core machines in parallel.

In each run, FOCAL always detected the 71 bugs in 1838.3 hours on average (262.6 hours on average per program), which consist of

- Fuzzing and function relevance measurement (FZ): 1 hour
- Target failure line identification (FLI): 553.4 hours
- Satisfiability check (SC): 77.8 hours
- Craig interpolant calculation (CC): 155.8 hours
- FS refinement (FSR): 1044.2 hours

In contrast, AFL-fast detected only 40 bugs with the same amount of time as that of FOCAL. KATCH and Mix-CCBSE detected only 34 and 25 bugs after spending 1.3 and 1.6 times larger amount of the execution time than FOCAL (i.e., total 2427.3 and 2924.0 hours), respectively. Since KATCH and Mix-CCBSE do not perform concolic testing in a compositional way, they need to explore large execution space to guide concolic testing to raise a failure at v. All bugs detected by these techniques were also detected by FOCAL.

5.2 RQ2: Effect of the Craig Interpolants in FSR

Table 3 shows that the Craig interpolants in the FSR improved bug detection ability. FOCAL detected 4.4 times more bugs (=71/16) than FOCAL$^{-\mathcal{I}}$.[5] Also, the table shows that the Craig interpolants-based FSR refines FSes effectively in terms of branch coverage (i.e., ϕ'_a covers a largely different set of branches than ϕ_a). $C(\phi_a)$ is a branch coverage of a achieved by a set of execution paths in ϕ_a and $C(\phi_a \cup \phi'_a)$ is a branch coverage of a achieved by a set of execution paths in ϕ_a or ϕ'_a.[6] Table 3 shows that FOCAL increases the branch coverage of each function by 17.3%p ($= C(\phi_a \cup \phi'_a) - C(\phi_a)$) by using FSR with the Craig interpolants (it generates 946.9 interpolants on average per program). In contrast, FOCAL$^{-\mathcal{I}}$ increases the branch coverage of each function by only 6.2%p.

The execution time of FOCAL$^{-\mathcal{I}}$ to build symbolic path formulas is shorter than that of FOCAL (i.e., 811.7 vs. 1277.8 which correspond to SC+CC+FSR) because FSR without Craig interpolants was not effective in resolving the conflicts and FOCAL$^{-\mathcal{I}}$ generates much fewer SPF than FOCAL (Sect. 6.1).

[5] FOCAL$^{-\mathcal{I}}$ always detected the 19 bugs in each of the 10 runs. The bugs detected by FOCAL$^{-\mathcal{I}}$ are also detected by FOCAL.

[6] For FOCAL which repeats the FSR step, ϕ'_a is the final refined FS (Sect. 3.5.4). For FOCAL$^{-\mathcal{I}}$, ϕ'_a is a FS refined for the same amount of the total refinement time spent by FOCAL.

Table 3: # of the detected target bugs, the time (hours) to build SPFs, and the effect of FSR of FOCAL$^{-\mathcal{I}}$ and FOCAL

Targets	FOCAL$^{-\mathcal{I}}$				FOCAL				
	# det. bugs	Time (h)	Branch Cov.(%) $C(\phi_a)$	Branch Cov.(%) $C(\phi_a \cup \phi'_a)$	#det. bugs	Time (h)	$\#\mathcal{I}$	Branch Cov. (%) $C(\phi_a)$	Branch Cov. (%) $C(\phi_a \cup \phi'_a)$
Bash	4	145.6	54.1	59.1	25	231.2	1170	54.1	66.8
Flex	1	69.5	55.3	62.0	4	103.7	531	55.3	73.8
Grep	2	129.8	59.1	67.9	5	231.7	1141	59.1	77.9
Gzip	2	61.6	56.5	61.0	4	99.4	571	56.5	72.0
Make	1	134.9	59.8	65.3	9	217.5	1198	59.8	78.7
Sed	1	75.1	52.9	60.1	3	123.0	570	52.9	71.8
Vim	5	195.3	48.2	53.3	21	271.2	1447	48.2	66.0
Sum	16	811.7	N/A	N/A	71	1277.8	6628	N/A	N/A
Avg.	2.3	116.0	55.1	61.3	10.1	182.5	946.9	55.1	72.4

Table 4: # of the detected bugs and the execution time to build SPFs of FOCAL^{-E}, FOCAL^{R}, and FOCAL.

Targets	FOCAL^{-E}		FOCAL^{R}		FOCAL	
	#det. bugs	Time (h)	#det. bugs	Time (h)	#det. bugs	Time (h)
Bash	10	902.9	7.4	238.8	25	231.2
Flex	2	173.4	1.2	97.6	4	103.7
Grep	2	617.4	2.6	314.9	5	231.7
Gzip	2	376.8	2.0	98.9	4	99.4
Make	3	712.9	2.8	331.6	9	217.5
Sed	1	468.3	1.4	203.1	3	123.0
Vim	12	1010.6	10.8	494.5	21	271.2
Sum	32	4262.3	28.2	1779.4	71	1277.8
Avg.	4.6	608.9	4.0	254.2	10.1	182.5

5.3 RQ3: Effect of the Extended Units on Bug Detection and Execution Time

The experiment results show that utilizing extended units contribute to high bug detection ability because FOCAL detected more than twice the number of bugs (71 bugs) than FOCAL^{-E} (32 bugs)[7] and FOCAL^{R} (28.2 bugs). Table 4 shows the numbers of the detected target bugs and the execution time to build SPFs of these techniques. Also, FOCAL spent only 1/3 of the time spent by FOCAL^{-E} (i.e., 1277.8 vs. 4262.3 hours) because FOCAL^{-E} identified 4.8 times more target failure lines than FOCAL (497 and 2402 target failure lines, respectively). FOCAL^{R} identified 1.7 times more target failure lines (i.e., 849.4 vs. 497) and spent 1.4 times larger amount of time (i.e., 1779.4 vs. 1277.8 hours) than FOCAL.

[7] FOCAL^{-E} always detected the 32 bugs in each of the 10 runs. The bugs detected by FOCAL^{-E} are also detected by FOCAL.

5.4 RQ4: New Crash Bug Detection

FOCAL detected 13 new crash bugs in the 12 target C programs in 782.5 hours on average [8]. We have reported the new bugs with crashing system-level test inputs to the developers of the target programs. Eight of them were confirmed by the developers and we have not received a response for the remaining five (we uploaded all responses in https://sites.google.com/view/focal-fse19).

For example of `libxml2-2.9.8`, FOCAL identified 87 target failure lines and detected two bugs in 124 hours: one buffer overflow bug (crashing at `HTMLparser.c:5408`) and one null pointer dereference bug (crashing at `xmlregexp.c:4349`). For example, to validate the buffer overflow bug of `libxml2-2.9.8`, FOCAL generates a pair of command line options (`--html` and `--push`) and an input file (a 765 bytes long xml file) as a test input.

6 DISCUSSION

6.1 Effectiveness of the Craig Interpolants Guided FSR for Bug Detection Ability

Since the amount of the execution time of FOCAL is proportional to a number of SPFs generated, reducing *non-validating* SPFs (i.e., satisfiable SPFs that correspond to the executions from `main` to v but those whose solutions do not validate the target failures) is important in detecting bugs effectively in a limited testing time. FOCAL uses under-approximate FSes to reduce non-validating SPFs because over-approximate FSes may lead compositional concolic testing to generate many non-validating SPFs.

FSR is crucial for FOCAL in detecting bug effectively because under-approximate FSes might not provide necessary execution contexts for ψ_v. With the help of the Craig interpolants as FS refining constraints, FOCAL generates 11.6 validating SPFs (each of which consists of 5.2 FSes on average) that reach `main` from the target failure lines and whose solutions validate the target failures per program on average, which is 4.5 times (=11.6/2.6) more than the validating SPFs generated by FOCAL$^{-\mathcal{I}}$. Consequently, FOCAL detects 4.4 times (=71/16) more bugs than FOCAL$^{-\mathcal{I}}$. Thus, we can conclude that FSR using the Craig interpolants as refining constraints significantly improves bug detection ability of FOCAL.

6.2 Effectiveness of Function Relevance-Based Extended Units for Execution Time and Bug Detection Ability

Realistic FSes are important for compositional concolic testing techniques to detect bugs effectively in a limited testing time. FOCAL uses function relevance-based extended units (Sect. 3.2 and Sect. 3.3) to obtain realistic FSes. Sect. 5.3 demonstrates that the function relevance based extended units contribute in reducing the execution time and improving bug detection ability of FOCAL.

First, since the amount of the execution time of FOCAL is proportional to a number of target failure lines, the extended units saved the execution time in a large degree by reducing (false) target failure lines. FOCAL and FOCAL^{-E} identified 67.1 and 313.9 target failure lines and spent 182.5 and 608.9 hours on average per program, respectively.

[8]AFl-fast, KATCH, and Mix-CCBSE detected 8, 7, and 5 crash bugs in 782.5, 1210.3, and 1610.9 hours, respectively. All bugs detected by them were detected by FOCAL.

Second, the extended units help FOCAL to reduce non-validating SPF generation. For example, FOCAL^{-E} generated 21.3 satisfiable SPFs that reach `main` from the target failure lines on average per program. But, only 5.4 test inputs obtained by solving these SPFs validate target failures (=25.4%=5.4/21.3). FOCAL generated 14.8 satisfiable SPFs that reach `main` from the target failure lines and 11.6 test inputs that validate the target failures (=78.4%=11.6/14.8) on average per program. [9] Thus, we can conclude that the extended units contribute to build SPFs that closely represent realistic system-level behaviors of a target program.

Third, the extended units also improve bug detection ability (i.e., 71 vs. 32 bugs detected by FOCAL and FOCAL^{-E}). This is because the FSes based on extended units (i.e., FOCAL) are more realistic and more compatible to combine to build SPFs than the ones based on single function (i.e., FOCAL^{-E}). For example, FOCAL generates 0.22 (= 14.8/67.1) satisfiable SPFs that reach `main` per target failure line while FOCAL^{-E} generates only 0.07 (=21.3/313.9) satisfiable SPFs that reach `main` per target failure line.

6.3 Comparison of the Directed Compositional Concolic Testing Techniques

We compare FOCAL with SMASH [16] and Alter [38] which are the most closely related work. Since the implementations of these techniques are not publicly available, we compare them in an analytic way. FOCAL uses an under-approximate summary of a function based on its extended unit, and then repeatedly refines the summary to cover program behaviors that are compatible with the target failures by using the Craig interpolants.

SMASH [16] generates an over-approximate summary (i.e., may-summary by predicate abstraction) and an under-approximate summary (i.e., must-summary by dynamic symbolic execution) of a function. It uses both summaries to prune the execution space that do not lead to a target failure. Unlike FOCAL, SMASH does not refine a must-summary and may fail to detect bugs. The experiment results with 69 device drivers [16] showed that SMASH is three times faster than a non-compositional may-must analysis technique DASH [4], but detects no more bugs than DASH.

Alter [38] explores symbolic space of a program in a goal-driven way with selectively composing over-approximate FSes. Alter uses Craig interpolants to check if the current search scope cannot have a solution towards a target failure. But FOCAL uses Craig interpolants to refine FSes to build satisfiable SPFs targeting v. The Alter paper does not show a *system-level* bug detection ability because Alter generates only test inputs to a public method nearest to a target failure (a method call distance from a nearest public method to a target failure is usually short), not to a program entry function (e.g., `main` in C programs). In contrast, FOCAL generates a test input that runs P from `main` to validate a target failure at v and, thus, fully demonstrates its bug detection ability as a system-level bug detection technique.

[9]A system-level test input obtained by solving a satisfiable SPF still may not validate/reproduce the target failure because ϕ_a may not represent real behaviors of a due to a's symbolic stubs that may return infeasible values.

7 RELATED WORK

7.1 Compositional Symbolic Analysis

SMART [14] generates a FS as a disjunction of function-wise symbolic path formulas (i.e., conjunctions of constraints over inputs and outputs of a function). However, since SMART does not refine FSes, the bug detection ability may be low. Godefroid et al. [14] reported only a case study on message parsing modules in oSIP without reporting bug detection ability. Anand et al. [2] extends SMART [14] by generating FSes as first-order formulas with uninterpreted functions. It refines a FS by refining uninterpreted functions on demand (not using Craig interpolants). However, unlike FOCAL that focuses on generating test inputs to validate target failures that were quickly identified by concolic unit testing, Anand et al. [2] may not detect bugs effectively in given limited time as it targets all uncovered code locations. Anand et al. [2] reported three small case studies on C# programs where the proposed technique detects only one more bug than Pex [39]. Qiu et al. [34] proposed a compositional symbolic execution for heap manipulating programs. They reported the execution time performed by the proposed approach to explore all feasible program paths, but did not report bug detection ability of the proposed approach.

For bounded model checking, FunFrog [36] and HiFrog [1] construct over-approximate FSes using Craig interpolants in a propositional logic and a quantifier-free linear real arithmetics with UF theory, respectively. Asadi et al. [3] proposed an on-demand FSR using different theories for bounded model checking. Unlike these techniques which use Craig interpolants to generate over-approximate FSes, FOCAL constructs and refines under-approximate FSes using Craig interpolants as refining constraints.

7.2 Directed Symbolic Analysis

Mix-CCBSE [30] combines backward call-chain exploration and forward shortest-distance guided symbolic execution to reach a given target code line, but without using FSes. Cilocnoc [11] combines symbolic backward execution and search-based concrete forward execution such that forward execution handles complex code which symbolic backward execution cannot analyze (e.g., external function calls or complex loops). Since Cilocnoc does not adopt compositional approach, it may suffer from search space explosion problem. Cilocnoc's bug detection ability is evaluated on seven toy programs by comparing time to reach the given goal line between Cilocnoc, jCUTE and Symbolic PathFinder.

BugRedux [20] generates a system-level test input that reproduces a failure from the system-level failed execution information such as a call stack dump or a call sequence obtained from the failed system-level execution. Similarly to BugRedux, Hercules [33] generates a system-level test input that reproduces a crash in real-world binary programs from a crash report (e.g., call stack dump, program state). First, Hercules identifies a crash condition from the crash report. Then, it performs symbolic execution and computes a minimal unsatisfiability core if a symbolic path formula σ conflicts with the crash condition. Hercules guides symbolic execution to resolve the conflict by negating every clause in σ that appears in the minimal unsatisfiability core. In contrast, FOCAL uses a Craig interpolant for FSR to resolve a conflict between ϕ_{a_k} and summaries of a current failure-context. [10] In addition, FOCAL can detect unknown failures where as BugRedux and Hercules cannot detect unknown failures, but reproduce failures only if information on a corresponding system-level execution is available.

Jaffar et al. [19] use a Craig interpolant-based search strategy to prune symbolic paths. Unlike FOCAL that utilizes compositional concolic testing, Jaffar et al.'s work analyzes a whole program by performing function inlining (i.e., limited scalability.) Jaffar et al. reported the execution time spent by the proposed technique to explore all feasible execution paths in relatively small target benchmark programs (i.e., SV-COMP12), but did not report bug detection ability. KATCH [31] uses a directed forward search strategy to cover changed portion of source code (i.e., a patch) effectively by using regression tests as initial tests of symbolic executions.

8 CONCLUSION

We present FOCAL which detects many bugs in programs without false alarms. A core idea of FOCAL is to effectively and quickly identify the target failures using concolic unit testing and focus to generate system-level tests that validate the target failures using compositional concolic testing with the Craig interpolants-based function summary refinement. The evaluation with the real-world C programs shows that FOCAL outperforms fuzzing (AFL-fast) and directed concolic testing (KLEE and Mix-CCBSE) techniques.

As future work, we will improve the FS composition strategy to prune the failure-contexts from which validating SPFs may not be generated. Also, we will improve the accuracy of function relevance metric by using machine learning techniques with various static and dynamic code features. Furthermore, we will expand the target domain of the compositional approach to invasive software testing [24] to reduce computational cost and mutation-based fault localization (MBFL) [18, 28, 32] to improve fault localization precision by generating more failing test inputs.

ACKNOWLEDGMENTS

This research has been supported by Next-Generation Information Computing Development Program through NRF funded by MSIT (NRF-2017M3C4A7068177 and NRF-2017M3C4A7068179), Basic Science Research Program through NRF funded by MSIT (NRF-2017R1C1B1008159 and NRF-2019R1A2B5B01069865), and Basic Science Research Program through NRF funded by MOE (NRF-2017R1D1A1B03035851).

REFERENCES

[1] Leonardo Alt, Sepideh Asadi, Hana Chockler, Karine Even Mendoza, Grigory Fedyukovich, Antti E. J. Hyvärinen, and Natasha Sharygina. 2017. HiFrog: SMT-based Function Summarization for Software Verification. In *Tools and Algorithms for the Construction and Analysis of Systems*, Axel Legay and Tiziana Margaria (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 207–213.

[2] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–381.

[10] Using Craig interpolants as path guiding constraints (FOCAL) may guide concolic testing to a target path more effectively than the minimal unsatisfiability core based constraints (Hercules). This is because the Craig interpolant is a goal-driven over-approximation of the failure conditions constructed so far (i.e., $\Phi_v(S_v^{k-1})$) and, thus, can serve as hints to guide concolic execution toward the target failures. In contrast, the minimal unsatisfiability core based constraints just prevent concolic execution from exploring the execution paths that cannot raise the target failures.

[3] Sepideh Asadi, Martin Blicha, Grigory Fedyukovich, Antti Hyv\"arinen, Karine Even-Mendoza, Natasha Sharygina, and Hana Chockler. 2018. Function Summarization Modulo Theories. In *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning (EPiC Series in Computing)*, Gilles Barthe, Geoff Sutcliffe, and Margus Veanes (Eds.), Vol. 57. EasyChair, 56–75. https://doi.org/10.29007/d3bt

[4] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. 2008. Proofs from Tests. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/1390630.1390634

[5] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[6] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259. https://doi.org/10.17487/RFC8259

[7] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08)*. IEEE Computer Society, Washington, DC, USA, 443–446. https://doi.org/10.1109/ASE.2008.69

[8] Arindam Chakrabarti and Patrice Godefroid. 2006. Software Partitioning for Effective Automated Unit Testing. In *Proceedings of the 6th International Conference on Embedded Software (EMSOFT '06)*. ACM, New York, NY, USA, 262–271.

[9] William Craig. 1957. Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. *The Journal of Symbolic Logic* 22, 3 (1957), 269–285. http://www.jstor.org/stable/2963594

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[11] Peter Dinges and Gul Agha. 2014. Targeted Test Input Generation Using Symbolic-concrete Backward Execution. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 31–36. https://doi.org/10.1145/2642937.2642951

[12] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Software Engineering* 10, 4 (Oct. 2005), 405–435.

[13] Gordon Fraser and Andrea Arcuri. 2013. 1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite. *Empirical Software Engineering* 20, 3 (2013), 611–639.

[14] Patrice Godefroid. 2007. Compositional Dynamic Test Generation. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 47–54. https://doi.org/10.1145/1190216.1190226

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223.

[16] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 2010. Compositional May-must Program Analysis: Unleashing the Power of Alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/1706299.1706307

[17] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. Search-Based System Testing: High Coverage, No False Alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, New York, NY, USA, 67–77.

[18] Shin Hong, Taehoon Kwak, Byeongcheol Lee, Yiru Jeon, Bongseok Ko, Yunho Kim, and Moonzoo Kim. 2017. MUSEUM: Debugging real-world multilingual programs using mutation analysis. *Information and Software Technology* 82 (2017), 80 – 95. https://doi.org/10.1016/j.infsof.2016.10.002

[19] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. 2013. Boosting Concolic Testing via Interpolation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 48–58. https://doi.org/10.1145/2491411.2491425

[20] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for In-house Debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 474–484. http://dl.acm.org/citation.cfm?id=2337223.2337279

[21] Moonzoo Kim, Yunho Kim, and Yunja Choi. 2012. Concolic testing of the multi-sector read operation for flash storage platform software. *Formal Aspects of Computing* 24, 3 (01 May 2012), 355–374. https://doi.org/10.1007/s00165-011-0200-9

[22] M. Kim, Y. Kim, and H. Kim. 2011. Comparative Study on Software Model Checkers as Unit Testing Tools: An Industrial Case Study. *IEEE Transactions on Software Engineering (TSE)* 37, 2 (March 2011), 146–160.

[23] Yunho Kim, Yunja Choi, and Moonzoo Kim. 2018. Precise Concolic Unit Testing of C Programs Using Extended Units and Symbolic Alarm Filtering. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/3180155.3180253

[24] Yunho Kim, Shin Hong, Bongseok Ko, Duy Loc Phan, and Moonzoo Kim. 2018. Invasive Software Testing: Mutating Target Programs to Diversify Test Exploration for High Test Coverage. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*.

[25] Yunho Kim and Moonzoo Kim. [n.d.]. CROWN: Concolic testing for Real-wOrld softWare aNalysis. http://github.com/swtv-kaist/CROWN Accessed: 2019-06-29.

[26] Yunho Kim, Youil Kim, Taeksu Kim, Gunwoo Lee, Yoonkyu Jang, and Moonzoo Kim. 2013. Automated Unit Testing of Large Industrial Embedded Software Using Concolic Testing. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Piscataway, NJ, USA, 519–528. https://doi.org/10.1109/ASE.2013.6693109

[27] Yunho Kim, Dongju Lee, Junki Baek, and Moonzoo Kim. 2019. Concolic Testing for High Test Coverage and Reduced Human Effort in Automotive Industry. In *International Conference on Software Engineering (ICSE) Software Engineering In Practice (SEIP) track*.

[28] Yunho Kim, Seokhyeon Mun, Shin Yoo, and Moonzoo Kim. To appear. Precise Learn-to-Rank Fault Localization using Dynamic and Static Features of Target Programs. *ACM Transactions on Software Engineering and Methodology* (To appear).

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 75–.

[30] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Proceedings of the 18th International Conference on Static Analysis (SAS'11)*. Springer-Verlag, Berlin, Heidelberg, 95–111. http://dl.acm.org/citation.cfm?id=2041552.2041563

[31] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage Testing of Software Patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 235–245. https://doi.org/10.1145/2491411.2491438

[32] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST '14)*. IEEE Computer Society, Washington, DC, USA, 153–162. https://doi.org/10.1109/ICST.2014.28

[33] Van-Thuan Pham, Wei Boon Ng, Konstantin Rubinov, and Abhik Roychoudhury. 2015. Hercules: Reproducing Crashes in Real-world Application Binaries. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 891–901. http://dl.acm.org/citation.cfm?id=2818754.2818862

[34] Rui Qiu, Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. 2015. Compositional Symbolic Execution with Memoized Replay. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 632–642. http://dl.acm.org/citation.cfm?id=2818754.2818832

[35] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, USA, 263–272.

[36] Ondrej Sery, Grigory Fedyukovich, and Natasha Sharygina. 2011. Interpolation-Based Function Summaries in Bounded Model Checking. In *Hardware and Software: Verification and Testing*, Kerstin Eder, João Lourenço, and Onn Shehory (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 160–175.

[37] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 201–211. https://doi.org/10.1109/ASE.2015.86

[38] Nishant Sinha, Nimit Singhania, Satish Chandra, and Manu Sridharan. 2012. Alternate and Learn: Finding Witnesses without Looking All over. In *Computer Aided Verification*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 599–615.

[39] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex: White Box Test Generation for .NET. In *Proceedings of the 2Nd International Conference on Tests and Proofs (TAP'08)*. Springer-Verlag, Berlin, Heidelberg, 134–153.

[40] Aaron Tomb, Guillaume Brat, and Willem Visser. 2007. Variably Interprocedural Program Analysis for Runtime Error Detection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA '07)*. ACM, New York, NY, USA, 97–107. https://doi.org/10.1145/1273463.1273478