Precise Concolic Unit Testing of C Programs using Extended Units and Symbolic Alarm Filtering

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

Yunja Choi  
School of Computer Science and Engineering, Kyungpook Natl. Univ.  
Daegu, South Korea  
yuchoi76@knu.ac.kr

Moonzoo Kim  
School of Computing, KAIST  
Daejeon, South Korea  
moonzoo@cs.kaist.ac.kr

ABSTRACT
Automated unit testing reduces manual effort to write unit test drivers/stubs and generate unit test inputs. However, automatically generated unit test drivers/stubs raise false alarms because they often over-approximate real contexts of a target function \( f \) and allow infeasible executions of \( f \). To solve this problem, we have developed a concolic unit testing technique CONBRIO. To provide realistic context to \( f \), it constructs an extended unit of \( f \) that consists of \( f \) and closely relevant functions to \( f \). Also, CONBRIO filters out a false alarm by checking feasibility of a corresponding symbolic execution path with regard to \( f \)’s symbolic calling contexts obtained by combining symbolic execution paths of \( f \)’s closely related predecessor functions.

In the experiments on the crash bugs of 15 real-world C programs, CONBRIO shows both high bug detection ability (i.e. 91.0% of the target bugs detected) and high precision (i.e. a true to false alarm ratio is 1:4.5). Also, CONBRIO detects 14 new bugs in 9 target C programs studied in papers on crash bug detection techniques.

CSC CONCEPTS
- Software and its engineering \( \rightarrow \) Software testing and debugging;

ACM Reference Format:

1 INTRODUCTION
Although unit testing is effective to detect SW bugs, field engineers have burden of manually generating test drivers/stubs and test inputs for each target unit. To reduce manual effort to generate test inputs, automated test generation has been applied (e.g., concolic testing have been applied to detect bugs in open source programs [2–4, 24, 28, 36] and industrial projects [6, 17, 22, 30, 42] at system-level).

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.

ICSE ’18, May 27-June 3, 2018, Gothenburg, Sweden  
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.  
ACM ISBN 978-1-4503-5638-1/18/05 ... $15.00  
https://doi.org/10.1145/3180155.3180253

Also, to reduce manual effort to generate unit test drivers/stubs, automated unit testing has been applied to open source programs [10, 14, 34] and large industrial SW [25].

A main drawback of the automated unit testing is a large number of false alarms raised by infeasible unit executions (i.e. unit executions that are infeasible at system-level). Infeasible unit executions occur generated due to inaccurate unit test drivers/stubs that over-approximate real contexts of a target unit (Sect. 2.4). This false alarm problem is a serious obstacle to apply automated unit testing in practice since field engineers would not like to spend time to manually filter out many false alarms.

To overcome this limitation, we have developed an automated unit testing framework CONBRIO (CONcolic unit testing with symbolic alarM filtering using symbolic calling conTexts) which operates in the following two stages:

1. To provide realistic context to a target function \( f \), CONBRIO constructs an extended unit of \( f \) that consists of \( f \) and closely relevant functions to \( f \) which can filter out infeasible unit executions caused by symbolic stubs. The relevance of a function \( g \) to \( f \) is measured by the degree of dependency of \( f \) on \( g \) (Sect. 3.2). Then, CONBRIO performs concolic execution of an extended unit of \( f \).

2. To filter out false alarms by checking feasibility of a corresponding symbolic unit execution of \( f \), CONBRIO generates symbolic calling context of \( f \) by combining symbolic paths of closely relevant predecessor functions of \( f \) in a static call graph.

As a result, CONBRIO detects bugs effectively and precisely because it enforces various and realistic executions of \( f \) through concolic execution of \( f \) with \( f \)’s realistic contexts (i.e., with the functions closely relevant to \( f \)) and accurately filters out false alarms using \( f \)’s symbolic calling contexts.

Note that it is important to construct an extended unit and symbolic calling context of \( f \) to contain only functions closely relevant to \( f \) since including more functions will enlarge symbolic search space and degrade unit testing effectiveness and efficiency. For example, at one extreme end, an extended unit may contain all successor functions of \( f \) and fail to detect bugs due to too large symbolic search space to explore. Also, symbolic calling context of \( f \) may contain symbolic execution paths of all predecessor functions of \( f \) up to main and fail to detect bugs. ¹

¹ Generated symbolic calling context may not represent all feasible calling context of \( f \) due to the limitation of symbolic execution. Thus, a symbolic calling context becomes more difficult to satisfy by adding symbolic execution paths of more predecessor functions of \( f \) (i.e., via logical \( \wedge \). See Sect. 3.5).

We have applied CONBRIO to 15 real-world C programs in SIR [11] and SPEC2006 [38] benchmarks and CONBRIO shows both
high bug detection ability (i.e., 91.0% of all target bugs detected) and high precision (i.e., a true to false alarm ratio is 1:4.5) which is more precise than the latest concolic unit testing techniques for C programs (e.g., 1:5.7 by UC-KLEE [34]). Also, CONBRIÖ detects 14 new bugs in the latest versions of the nine target C programs studied in other papers on crash bug detection techniques.

The contributions of this paper are as follows:

* CONBRIÖ achieves both high bug detection ability (91.0% of the target bugs detected) and high precision (false alarm ratio is 1:4.5) based on the two core ideas: 1) building and utilizing contexts of a target function explicitly based on relevance of functions measured by a function dependency metric, 2) a new alarm filtering strategy that constructs symbolic calling contexts compositionally and utilizes them to check feasibility of a violating unit execution.

* The extensive empirical evaluation on both bug detection ability and precision of CONBRIÖ and the other concolic unit testing techniques on the 15 real-world C programs supports researchers and practitioners to learn the pros and cons of the related techniques (Sect. 4–5).

* By applying CONBRIÖ, we have detected and reported 14 new crash bugs in the latest versions of the 9 target programs that were studied in other papers on crash bug detection techniques (Sect. 5.5).

* We have made the real-world crash bug data of the C benchmark programs publicly available, which were collected and organized after examining the bug reports of the last 12–24 years (http://swtv.kaist.ac.kr/tools/conbrio), so that researchers can use them for various testing research purposes (Sect. 4.2.1).

The remainder of the paper is as follows. Section 2 explains the background of automated concolic unit testing. Section 3 describes the detail of CONBRIÖ. Section 4 explains the experiment setup to evaluate CONBRIÖ compared to other techniques. Section 5 reports the experiment results. Section 6 discusses related work and Section 7 concludes the paper with future work.

2 BACKGROUND

2.1 Preliminary

Unit testing uses drivers and stubs (or mock objects) to test a target function in isolation (i.e., without the rest of a target program). Suppose that a target function under test \( f \) takes \( n \) arguments \( a_1, ..., a_n \) and accesses \( m \) global variables \( v_1, ..., v_m \), and directly calls \( l \) other functions \( g_1, ..., g_l \). To enforce diverse test executions of \( f \), a tester develops various unit test drivers \( d_r \)'s \( s \) each of which generates argument values \( a^1, ..., a^r \), \( p^1, ..., p^r \) and finally invokes \( f \) with these input values. Also, a tester builds stub functions \( s_{g1}^1, ..., s_{gl}^l \) to replace \( g_1, ..., g_l \). Also, test drivers/stubs should satisfy constraints on the interface between \( f \) and the rest of a target program to avoid infeasible unit test executions of \( f \).

2.2 Concolic Unit Test Driver/Stub Generation

For each target function \( f \), a concolic unit testing technique automatically generates symbolic stubs and a symbolic unit test driver. Symbolic stubs simply return symbolic values (without updating global variables and output parameters for simplicity) and a symbolic driver invokes \( f \) after assigning symbolic values to the input variables of \( f \) according to their types as follows:

* **primitive types**: primitive variables are directly assigned with primitive symbolic values of the corresponding types.
* **array types**: each array element is assigned with a symbolic variable according to the type of the array element (for a large array, only the first \( n \) elements are assigned with symbolic values where \( n \) is given by a user).
* **pointer types**: for a pointer variable \( ptr \) pointing to a variable of a type \( T \), a driver allocates memory whose size is equal to the size of \( T \) and assigns the address of the allocated memory to \( ptr \) (i.e., \( ptr = malloc(sizeof(T)) \)). Then, a driver assigns \( *ptr \) with a symbolic value of type \( T \). If a size of \( T \) is not known (e.g., FILE in standard C library), NULL is assigned to \( ptr \). If there exists a pointer variable \( ptr2 \) pointing to a symbolic variable of the same type \( T \), a driver assigns \( ptr2 \) to \( ptr \).
* **structure types**: a unit test driver specifies all fields of struct variable \( s \) as symbolic variables recursively (i.e., if \( s \) contains struct variable \( t \), a unit test driver specifies the fields of \( t \) as symbolic too).

A limitation of this approach is that the drivers and stubs often over-approximate the real environment of \( f \) and allow infeasible unit executions (i.e., executions of \( f \) which are not feasible at system-level) that may raise false alarms.

2.3 Insertion of Assertions Targeting Crash Bugs

Concolic unit testing techniques aim to detect crashes/run-time failures such as null-pointer dereference (NPD), array index out-of-bounds (OOB), and divide-by-zero (DBZ) as well as violations of user-given assertions. They often focus on crashes because user-given assertions are usually not available in real-world programs.

Concolic unit testing techniques insert \( assert(exp) \) into \( f \) where \( exp \) specifies a condition to avoid crashes (e.g., \( denominator \neq 0 \) to avoid DBZ). Because of \( assert(exp) \) in \( f \), concolic testing tries to generate a test input with which \( f \) makes \( exp \) false and increases a chance to detect crash bugs.

2.4 Example of False Alarm

Figure 1 shows a target program with a target function \( f \) under test (lines 10–16), main calls \( a1 \) if the first parameter \( x \) of \( \text{main} \) is greater than 0 or calls \( a2 \), otherwise (line 3). \( a1 \) and \( a2 \) call \( b \) at line 5 and line 6, respectively, and \( b \) calls \( f \) at line 7. \( f \) takes an integer parameter \( x \) and calls \( g(x) \) (line 12) (a sanity check function for accessing an array through an index \( x \)) and \( h(x) \) (line 15). A concolic unit testing technique generates a unit test driver \( \text{driver}_f \) and symbolic stubs \( \text{stub}_g \) and \( \text{stub}_h \) for \( f \). Also, it modifies \( f \) to call \( \text{stub}_g \) and \( \text{stub}_h \) instead of \( g \) and \( h \) respectively (see the comments at line 12 and line 15) and inserts an OOB assertion at line 13.

Figure 2 shows a unit test driver and stubs for \( f \). \( \text{driver}_f \) invokes \( f \) with a symbolic argument \( \text{arg1} \) (lines 2–3) where \( \text{int arg1} = \text{SymbInt()} \) sets \( \text{arg1} \) as a symbolic integer value (line 2). \( \text{stub}_g \)
Precise Concolic Unit Testing of C Programs using
Extended Units and Symbolic Alarm Filtering

1. It checks function relevance by calculating 
   \( \text{stub}_g \) values (lines 5–7 and lines 9–11 respectively). Concolic execution
   14: `result = array[x];`
   13: `// => assert(0<=x && x<5);`
   11: `int array[5] = {1,3,5,7,9}, result;`
   10: `int f(int x){`

3 CONBRIO TECHNIQUE

Section 3.3 describes more detail.

4. CONBRIO applies concolic testing to an extended unit of \( f \) to explore
diverse and realistic target unit test executions. During concolic
execution, it builds a symbolic path formula \( \sigma_{\Delta} \), that repre-

5. It filters out an alarm raised at \( v_i \) by checking the feasibility of
   \( \sigma_{\Delta_{\text{inv}}} \) with regard to \( f \)’s calling contexts (see Step 3). For this
   purpose, CONBRIO constructs \( f \)’s symbolic calling context formulas
   \( \Sigma_{\text{ctx}}(f,k) \) and uses an SMT solver to check satis-

3.1 Obtaining Function Call Profile from
System Test Executions

CONBRIO executes a target program with given system test cases
and obtains function call profiles. For example, suppose that a
target program in Fig. 1 has three system test cases to
\( \text{main}(x,y) \): (-1,1), (1,1), and (5,1). Then, the function call profiles are obtained
as follows: \{main\( \rightarrow \text{a2},\) \( a2\rightarrow b,\) \( b\rightarrow f,\) \( f\rightarrow g) \} with (-1,1), \{main\( \rightarrow \text{a1},\) \( a1\rightarrow b,\) \( b\rightarrow f,\) \( f\rightarrow g,\) \( f\rightarrow h) \} with (5,1).

3.2 Computing Dependency of a Target
Function on Other Functions

Suppose that a program has a target function \( f \) and other function
\( g \) and it has \( n_f \) system test executions that invokes \( f \). Based on
function call profiles, we compute dependency of \( f \) on \( g \) as \( p(g|f) \).

With a given dependency threshold \( \tau \), we consider \( f \) has a high
dependency on \( g \) if \( p(g|f) \geq \tau \).

3. Based on the calculated dependency of \( f \) on other functions,
   • It constructs an extended unit of \( f \) that contains \( f, \) \( f \)'s successor
     functions in a static function call graph on which \( f \) has high
dependency, and symbolic stubs.
   • It identifies calling contexts of \( f \) each of which is a maximal
call path \( a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow f \) in a static function call graph
     such that \( f \) has high dependency on all \( a_d \)'s. We use \( \text{ctx}(f,k) \)
to indicate \( k \)th calling context of \( f \).

\[
\begin{align*}
\text{driver}_{f}(): & \quad \text{int} \ arg1 = \text{SYM}_{\text{int}}(); \\
& \quad f(arg1); \\
\text{stub}_g() & \quad \text{int} \ stub\_g(int \ x) \\
\text{stub}_h() & \quad \text{int} \ stub\_h(int \ x) \\
\end{align*}
\]

\[
\begin{align*}
\text{array}[0] &= \{1,3,5,7,9\}, \text{result}; \\
\text{result} &= \text{array} \times x; \\
\text{return result} & \quad \text{return result}; \\
\end{align*}
\]

and \( \text{stub}_h \) return symbolic integer values as \( g \) \( \text{and} \) \( h \) return integer values (lines 5–7 and lines 9–11 respectively). Concolic execution
of \( \text{driver}_{f}() \) violates the OOB assertion at line 13 of \( f \) if a unit test
execution satisfies the following two conditions:

- a symbolic argument \( \text{arg1} \) to \( f \) (line 3 of \( \text{driver}_{f}() \)) is larger than
  or equal to the size of \( \text{array} \) (e.g. \( \text{arg1} \) is 5)
- \( \text{stub}_g \) returns a non-zero value (e.g. 1)

However, an alarm raised in such unit test execution is a false
alarm because such unit test execution of \( f \) is infeasible with the
real target program where \( g \) is invoked (g returns 0 if \( \text{arg1} \geq 5 \)
(line 18 of Fig 1) unlike \( \text{stub}_g \)). In other words, a concolic unit
testing technique can raise a false alarm if it generates unit test
drivers/stubs different from real environment of \( f \) which consist of
\( \text{main}, \text{a1}, \text{a2}, \text{b}, \text{g}, \) and \( h \).

3 CONBRIO TECHNIQUE

Figure 3 shows the overall process of CONBRIO as follows:
1. CONBRIO receives source code of a target program, a list of target
functions to test, and system test cases of the target program as
inputs. CONBRIO obtains function call profiles from the system

2. It checks function relevance by calculating dependency of a tar-
get function \( f \) on other function \( g \) using conditional probability
\( p(g|f) \) based on the observed function call profiles (Section 3.2).

\[
\begin{align*}
\text{array}[0] &= \{1,3,5,7,9\}, \text{result}; \\
\text{result} &= \text{array} \times x; \\
\text{return result} & \quad \text{return result}; \\
\end{align*}
\]
in Fig. 1. Based on the profiles, we calculated dependency of \( f \) on other functions as follows:

- \( p(\text{main}|f) = 1.00 \) (\( \frac{n_f}{n_\text{main}} = \frac{3}{3} \))
- \( p(a1|f) = 0.66 \) (\( \frac{n_f}{n_{a1}} = \frac{2}{3} \))
- \( p(a2|f) = 0.33 \) (\( \frac{n_f}{n_{a2}} = \frac{1}{3} \))
- \( p(b|f) = 1.00 \) (\( \frac{n_f}{n_b} = 1 \))
- \( p(g|f) = 1.00 \) (\( \frac{n_f}{n_g} = 1 \))
- \( p(h|f) = 0.33 \) (\( \frac{n_f}{n_h} = \frac{1}{3} \))

### 3.3 Constructing Extended Unit and Calling Contexts

Given a static call graph \( G(V,E) \) of a target program (Def.1), a target function \( f \)'s dependency on other functions (i.e., \( p(g|f) \)), and a dependency threshold \( \tau \), CONBRIO constructs an extended unit of \( f \) that consists of \( f \) and \( f \)'s closely relevant successor functions and calling contexts of \( f \).

**Definition 1.** A static call graph \( G(V,E) \) is a directed graph where \( V \) is a set of nodes representing functions in a program and \( E \) is a relation \( V \times V \). Each edge \((a,b) \in E \) indicates that \( a \) directly calls \( b \). We call a node \( p \) a predecessor of \( f \) if there exists a path from \( p \) to \( f \). We call a node \( s \) a successor of \( f \) if there exists a path from \( f \) to \( s \).

For example, Step 3 of Fig. 3 shows how CONBRIO constructs an extended unit of \( f \) and a calling context of \( f \) for a program in Fig. 1. Given a static call graph whose nodes are labelled with dependency of \( f \), CONBRIO constructs an extended unit of \( f \) that contains \( f \) and \( g \) since \( f \) has high dependency on \( g \), but not \( h \) (i.e., \( p(g|f) \geq \tau \) but \( p(h|f) < \tau \) where \( \tau = 0.7 \)). Finally, \( \text{driver}_f \) invokes an extended unit of \( f \) with symbolic inputs. Note that CONBRIO does not raise a false alarm in this example unlike concolic unit testing in Sect. 2.4 because an extended unit provides realistic environment to \( f \) by using \( g \) which is closely relevant to \( f \). Also, CONBRIO builds a calling context of \( f \) as \( b \rightarrow f \) since \( f \) has high dependency on \( b \), but not \( a1 \) nor \( a2 \) (i.e., \( p(b|f) \geq \tau \) but \( p(a1|f), p(a2|f) < \tau \)).

#### 3.3.1 Constructing Extended Unit

For each target function \( f \), CONBRIO constructs an extended unit that contains \( f \) and \( f \)'s successor functions \( g \) such that \( f \) has high dependency on all function nodes in a call path from \( f \) to \( g \) in a static function call graph (i.e., for all nodes \( n_i \) between \( f \) and \( g \), \( p(n_i|f) \geq \tau \)). A unit test driver sets all arguments and all global variables accessed by the extended unit of \( f \) as symbolic inputs as described in Sect. 2 and invokes \( f \).

For example, Fig. 4 shows a static call graph whose nodes are labelled with dependency of a target function \( f \). Fig. 4 shows that an extended unit of \( f \) (marked with black dashed line at the bottom) consists of \( f \), \( n12 \), \( n13 \), and \( n14 \) functions on which \( f \) has high dependency (i.e., \( p(n12|f), p(n13|f), p(n14|f) \geq \tau = 0.7 \)).
In addition, as a false alarm reduction heuristic, CONBRIO adds `SYM_assume(expr)` at the beginning of f’s extended unit where `expr` represents possible value ranges of symbolic input variables (which are obtained by applying a static value range analyzer [33] to an entire target program code). If an input value is not in the estimated range, a current test execution immediately terminates without raising any alarms and CONBRIO continues to a next test execution. As another heuristic, CONBRIO constructs an extended unit to keep consistency between a pointer input variable to dynamically allocated memory and its size variable by figuring out such relation between input variables based on the variable names.

DEFINITION 2. An ith calling context of f ∈ V (saying ctx(f,i)) is a maximal call path a1 → a2 → ... → f in a static call graph G(V,E) satisfying the following conditions:

- a1 is a predecessor of f
- for all ai in ctx(f,i), p(ai,f) ≥ \tau
- there exists no other calling context of f that contains ctx(f,i) as its sub path (i.e., ctx(f,i) is maximal).

CONBRIO generates a calling context by traversing a static call graph from f in a reverse direction until it reaches a node labelled with low dependency of f. For example, Fig. 4 shows two calling contexts of f: ctx(f,1) and ctx(f,2). ctx(f,1) is a call path from n5 to f (see the blue dotted line in the left part) where p(n5,f) = p(n8,f) = 0.8 > \tau = 0.7 and p(n1,f) = 0.5, p(n2,f) = 0.6. Thus, ctx(f,1) = n5 → n8 → f. Similarly, ctx(f,2) = n3 → n6 → n9 → f.

3.4 Concolic Testing to Generate Violating Symbolic Path Formulas

CONBRIO applies concolic testing to an extended unit to explore diverse and realistic executions of f. During concolic execution, it obtains a set of symbolic execution path formulas SE_f, and records a symbolic path formula σ_{f,v_i} that violates an assertion v_i in f (j is an index to a symbolic path formula violating v_i since there can be multiple such symbolic path formulas). We use σ_{f,v_i} to denote V_j σ_{f(v_i,j)}.

To focus on f, CONBRIO modifies DFS search strategies by using a priority queue for branch conditions of f and a normal queue for those of the other functions in an extended unit of f (e.g., g in Fig. 1). CONBRIO explores various behaviors of f first by negating branch conditions in a priority queue first (branch conditions in a normal queue are negated when the priority queue is empty).

3.5 Alarm Filtering by Checking Satisfiability of f’s Violating Symbolic Path Formula σ_{f,v_i} with f’s Symbolic Calling Context Formula

To filter out false alarms raised at v_i in f, CONBRIO checks the feasibility of σ_{f,v_i} with regard to f’s calling contexts (see Sect. 3.3.2). For this purpose, CONBRIO constructs \Pi_{ctx(f,k)} which is a 4th symbolic calling context of f and checks satisfiability of σ_{f,v_i} ∧ \Pi_{ctx(f,k)} using a SMT solver. \Pi_{ctx(f,k)} is constructed as follows (see Fig. 5):

- For each function a’ in a calling context of f (i.e., ctx(f,k)), CONBRIO obtains SE_{a’}, which is a set of symbolic execution path formulas of a’.
- If σ_i is a prefix of σ ∈ SE_{a’} such that σ contains inversion of a’+1 and σ’ does not contain a suffix of σ after the inversion.

For example, for ctx(f,1) in Fig. 4, Fig. 5 shows that Slice(SE_{n5}, n8) has two symbolic path formulas that call n8: \sigma_{n5(n8,1)} and \sigma_{n5(n8,2)} (shown as thick blue arrows) where µ_{n8,y} is 2nd symbolic path formula of a function x that terminates immediately after calling a function y. Slice(SE_{n8,f}) also has two symbolic path formulas that call f: \sigma_{n8(f,1)} and \sigma_{n8(f,2)}.

CONBRIO obtains symbolic calling context formula of f with ctx(f,k) (i.e., \Pi_{ctx(f,k)}) by combining sets of sliced symbolic execution path formulas of a’ (i.e., Slice(SE_{a’}, a^2), a^2 (i.e., Slice(SE_{a’}, a^3)), ... of ctx(f,k) until reaching f using logical conjunction. Thus, \Pi_{ctx(f,k)} with ctx(f,k) = a^1 → a^2 → ... → f is defined as follows:

\Pi_{ctx(f,k)} = \sigma_{\Pi}(f,k) \lor (\sigma_{\Pi} Slice(SE_{a’}, a^2+1))

For example, Fig. 5 shows the \Pi_{ctx(f,k)} of ctx(f,1) = n5 → n8 → f in Fig. 4 as follows (see thick blue arrows representing \sigma_{n5(n8,1)}, \sigma_{n5(n8,2)}, \sigma_{n8(f,1)} and \sigma_{n8(f,2)}):

\sigma_{\Pi}(f,k) = \Pi_{ctx(f,k)} = \Pi_{ctx(f,1)} \lor (\sigma_{\Pi} Slice(SE_{a’}, a^2+1))

Finally, CONBRIO applies a SMT solver to σ_{f,v_i} ∧ \Pi_{ctx(f,k)} for every symbolic calling context of f. If a result is UNSAT for all calling contexts (i.e., there exists no feasible execution in any calling contexts of f to make σ_{f,v_i} feasible), a target alarm is considered as false and ignored. Otherwise (i.e., a result is SAT with at least one
calling context), a corresponding alarm is reported as a violation of $v_i$ in $f$.

3.6 Implementation

We have implemented CONBRIO in 5,000 lines of C++ code using Clang/LLVM-3.4 [26]. CONBRIO uses CROWN [1] for concolic testing and LLVM-based static variable range analyzer [33] to compute the possible ranges of variables. CROWN (Concolic testing for Real-world Software Analysis) is a lightweight instrumentation-based concolic testing tool to generate concrete test inputs for real-world C programs (available at http://github.com/swtv-kaist/CROWN). It supports complex C features such as bitwise operators, floating point arithmetic, bitfields and so on.

4 EXPERIMENT SETUP

We have designed five research questions to evaluate bug detection ability and precision of CONBRIO and compare CONBRIO with other concolic unit testing techniques on 15 real-world C programs. Note that it is important to evaluate bug detection ability and precision together because of a trade-off between them (i.e., a technique may improve bug detection ability at the cost of precision or vice versa). Also, we applied CONBRIO to the latest versions of the nine C programs studied in other papers on crash bug detection techniques.

4.1 Research Questions

RQ1. Bug Detection Ability: How many crash bugs among the target crash bugs does CONBRIO detect, compared to the other concolic unit testing techniques?

RQ2. Bug Detection Precision: How much is a false alarm ratio of CONBRIO, compared to the other techniques?

RQ3. Effectiveness of the Symbolic Alarm Filtering: How much does the alarm filtering strategy using symbolic calling contexts affect a number of target bugs detected and a false alarm ratio?

RQ4. Effect of the Function Selection Strategy on Bug Detection Ability and Precision: How much does the function selection strategy based on the function relevance metric affect a number of target bugs detected and a false alarm ratio, compared to a strategy based on static call graph distance?

RQ5. Effectiveness of Detecting New Crash Bugs: How many new crash bugs does CONBRIO detect?

4.2 Target Bugs and Programs

We target crash bugs described in Section 2.3 by inserting corresponding crash assertions in target programs because crash bugs are serious problems and CONBRIO can automatically insert such assertions without user-given test oracles, which are rarely available in target programs. We use two benchmarks: known crash bug benchmark for RQ1 to RQ4 and unknown crash bug benchmark for RQ5 (available at http://swtv.kaist.ac.kr/tools/conbrio).

4.2.1 Known Crash Bug Benchmark. The known crash bug benchmark consists of all C programs in SIR [11] (except Siemens programs and space which do not have available bug-fix histories) and SPEC2006 integer benchmarks (except mcf-1.2 which has only one system test case). We target the crash bugs of the benchmark programs that satisfy the following conditions:

- crash bugs that exist in a target program version and have been confirmed by original developers through bug-fix commits since the release of a target program version (e.g., Dec 1996 for bash-2.0) until April 2017
- crash bugs that can be detected by unit testing (i.e., both buggy statement(s) reported in a bug-fix commit and violated assertion(s) are located in a same target function)

Table 1: Target programs and bugs for RQ1 to RQ4

<table>
<thead>
<tr>
<th>Target programs and versions</th>
<th>Lines</th>
<th># of func.</th>
<th># of sys. test cases</th>
<th>Branch cov. (%)</th>
<th>Func. cov. (%)</th>
<th># of target bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bash-2.0</td>
<td>32714</td>
<td>1214</td>
<td>1100</td>
<td>46.2</td>
<td>89.0</td>
<td>6</td>
</tr>
<tr>
<td>Flex-2.4.3</td>
<td>7471</td>
<td>147</td>
<td>567</td>
<td>45.7</td>
<td>93.9</td>
<td>9</td>
</tr>
<tr>
<td>Grep-2.0</td>
<td>5956</td>
<td>132</td>
<td>809</td>
<td>50.3</td>
<td>94.7</td>
<td>5</td>
</tr>
<tr>
<td>Grep-1.0.7</td>
<td>3054</td>
<td>82</td>
<td>214</td>
<td>55.8</td>
<td>87.8</td>
<td>2</td>
</tr>
<tr>
<td>Make-3.75</td>
<td>28715</td>
<td>555</td>
<td>1043</td>
<td>64.5</td>
<td>87.9</td>
<td>3</td>
</tr>
<tr>
<td>Sed-1.17</td>
<td>4085</td>
<td>73</td>
<td>360</td>
<td>47.3</td>
<td>87.7</td>
<td>2</td>
</tr>
<tr>
<td>Vim-5.0</td>
<td>66209</td>
<td>1749</td>
<td>975</td>
<td>35.8</td>
<td>91.0</td>
<td>6</td>
</tr>
<tr>
<td>Perl-5.8.7</td>
<td>79873</td>
<td>2240</td>
<td>1201</td>
<td>52.3</td>
<td>95.0</td>
<td>6</td>
</tr>
<tr>
<td>Bzip2-1.0.3</td>
<td>4737</td>
<td>114</td>
<td>6</td>
<td>67.4</td>
<td>93.9</td>
<td>2</td>
</tr>
<tr>
<td>Gcc-3.2</td>
<td>342611</td>
<td>5353</td>
<td>9</td>
<td>43.7</td>
<td>96.2</td>
<td>15</td>
</tr>
<tr>
<td>Gobmk-3.3.14</td>
<td>154583</td>
<td>2682</td>
<td>1354</td>
<td>65.2</td>
<td>92.0</td>
<td>5</td>
</tr>
<tr>
<td>Hummer-2.0.42</td>
<td>35992</td>
<td>539</td>
<td>4</td>
<td>75.6</td>
<td>94.1</td>
<td>3</td>
</tr>
<tr>
<td>Sjeng-11.2</td>
<td>10146</td>
<td>144</td>
<td>3</td>
<td>77.9</td>
<td>91.7</td>
<td>2</td>
</tr>
<tr>
<td>Libquantum-0.2.4</td>
<td>2255</td>
<td>101</td>
<td>3</td>
<td>68.5</td>
<td>93.1</td>
<td>3</td>
</tr>
<tr>
<td>H264ref-9.3</td>
<td>51578</td>
<td>590</td>
<td>6</td>
<td>63.6</td>
<td>88.0</td>
<td>5</td>
</tr>
<tr>
<td>Sum</td>
<td>829929</td>
<td>15915</td>
<td>7654</td>
<td>N/A</td>
<td>N/A</td>
<td>67</td>
</tr>
<tr>
<td>Average</td>
<td>55328.6</td>
<td>1061.0</td>
<td>510.3</td>
<td>57.3</td>
<td>91.7</td>
<td>4.5</td>
</tr>
</tbody>
</table>

RQ1: Bug Detection Ability: How many crash bugs among the target crash bugs does CONBRIO detect, compared to the other techniques?

RQ2: Bug Detection Precision: How much is a false alarm ratio of CONBRIO, compared to the other techniques?

RQ3: Effectiveness of the Symbolic Alarm Filtering: How much does the alarm filtering strategy using symbolic calling contexts affect a number of target bugs detected and a false alarm ratio?

RQ4: Effect of the Function Selection Strategy on Bug Detection Ability and Precision: How much does the function selection strategy based on the function relevance metric affect a number of target bugs detected and a false alarm ratio, compared to a strategy based on static call graph distance?

RQ5: Effectiveness of Detecting New Crash Bugs: How many new crash bugs does CONBRIO detect?
Table 1 describes 15 target programs including their sizes (in LoC including comments and empty lines), a number of functions to test, a number of system test cases used, branch coverage and function coverage achieved by the system test cases, and a number of the target crash bugs. For all target programs, we used all system test cases provided in the benchmarks. Each target program has two to 15 target crash bugs (4.5 on average). Note that no system test case detects a target bug.

For example, we have reviewed 28 bug-fix commits reported since the release of vim-5.0 on Feb 1998 until April 2017. 11 among them report crash bugs existing in vim-5.0. Among the 11 crash bugs, unit testing can detect six of them, which we target for vim-5.0 (see the eighth row of the table).

Table 2: Target programs for RQ5

<table>
<thead>
<tr>
<th>Target programs and versions</th>
<th>Lines</th>
<th># of func.</th>
<th># of sys. test cases</th>
<th>Branch cov. (%)</th>
<th>Func. cov. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcm2ps-8.13.9</td>
<td>36595</td>
<td>499</td>
<td>12</td>
<td>74.0</td>
<td>93.0</td>
</tr>
<tr>
<td>autotrace-0.31.1</td>
<td>18495</td>
<td>343</td>
<td>5</td>
<td>69.3</td>
<td>84.8</td>
</tr>
<tr>
<td>bib2xml-5.11</td>
<td>77216</td>
<td>1032</td>
<td>24</td>
<td>73.2</td>
<td>93.0</td>
</tr>
<tr>
<td>catddie-0.14</td>
<td>12693</td>
<td>187</td>
<td>7</td>
<td>53.0</td>
<td>81.8</td>
</tr>
<tr>
<td>eog-3.14.1</td>
<td>43463</td>
<td>605</td>
<td>42</td>
<td>73.3</td>
<td>81.0</td>
</tr>
<tr>
<td>gif2png-2.5.11</td>
<td>4058</td>
<td>76</td>
<td>2</td>
<td>60.1</td>
<td>81.6</td>
</tr>
<tr>
<td>jpegtran-1.3.1</td>
<td>51828</td>
<td>817</td>
<td>33</td>
<td>72.0</td>
<td>84.9</td>
</tr>
<tr>
<td>mp3gain-1.5.2</td>
<td>5786</td>
<td>100</td>
<td>3</td>
<td>53.7</td>
<td>86.0</td>
</tr>
<tr>
<td>spdf-3.03</td>
<td>22309</td>
<td>381</td>
<td>13</td>
<td>54.9</td>
<td>81.9</td>
</tr>
<tr>
<td>Sum</td>
<td>272443</td>
<td>4040</td>
<td>141</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Average</td>
<td>30271.4</td>
<td>448.9</td>
<td>15.7</td>
<td>64.8</td>
<td>85.3</td>
</tr>
</tbody>
</table>

Table 2: Target programs for RQ5

4.4 Measurement

We consider a target bug is detected if a unit test execution that violates an assertion covers one of the buggy statements in a target unit. To identify the buggy statements, we have manually analyzed all crash bug-fix commits of all subsequent releases of the target program versions in SIR and SPEC2006 benchmarks. We consider that a statement \( s \) of a target program is a buggy statement if it corresponds to the changed/fixed statements in a crash bug-fix commit.

We analyze alarms reported by the alarm filtering strategy (Sect. 3.5). For true alarms, we count a number of violated assert statements which satisfy the following conditions:

- There exists a unit test execution \( \sigma_f \), that covers a buggy statement and violates an assert statement in a target function \( f \).
- We can confirm that \( \sigma_f \) is feasible at system level by manually creating a system-level test that includes \( \sigma_f \) and violates the assert statement (we compared execution traces of \( \sigma_f \) and a corresponding system test using gdb).

We consider all other alarms as false ones.

4.5 Testbed Setting

For SUT, call-graph distance techniques, and CONBRIO, we set the timeout of concolic testing (Step 4 in Fig. 3) as 180 seconds per a target function. After test generation terminates, call-graph distance techniques and CONBRIO performs the false alarm filtering task (Step 5 in Fig. 3). We set a function dependency threshold \( \tau \) as 0.7.

Since the experiment scale is large (i.e., targeting 15,915 functions for the known crash bugs and 4,040 functions for the unknown crash bugs), the experiments were performed on 100 machines each of which is equipped with Intel quad-core i5 4670K and 8GB ram, running Ubuntu 14.04.2 64 bit version. We run four concolic unit test runs on a machine in parallel.

4.3 Concolic Unit Testing Techniques to Compare

We have compared CONBRIO with the following concolic unit testing techniques:

- **Symbolic unit testing (SUT):** It generates a symbolic unit testing driver with symbolic arguments to a target function \( f \) and symbolic global variables without any constraints on the symbolic values, as described in Sect. 2.2. Also, SUT uses symbolic stubs to replace all functions called by \( f \).
- **Static call-graph distance techniques:** It constructs an extended unit to include all successor functions of \( f \) within a certain distance bound from \( f \) in a static function call graph. Also, a calling context of \( f \) contains predecessor functions of \( f \) within a certain distance bound from \( f \). We use distance bounds 3, 6 and 9. SUT corresponds to a static call-graph distance technique with a distance bound 0.

SUT uses DFS as a concolic search strategy. Call-graph distance techniques and CONBRIO use the modified DFS (Sect. 3.4). These unit testing techniques have been implemented in 1,000 lines of C++ code using CROWN [1].
4.6 Threats to Validity

A threat to external validity is the representativeness of our target programs. But we expect that this threat is limited since the target programs are widely used real-world ones and tested by many other researchers. Another threat to external validity is the possible bias of the system tests we used to obtain dependency between functions. We tried to reduce this threat by utilizing all available system test cases in the benchmarks.

A threat to internal validity is possible faults in the implementation of the concolic unit testing techniques we studied. To address this threat, we extensively tested our implementation. A threat to construct validity is the use of the crash bugs that were fixed by the bug-fix commits reported so far (i.e., the target programs may have unknown/unreported crash bugs which we do not count). We target crash bugs confirmed by the developers through the bug-fix commits because it would require too much effort to manually validate numerous alarms without confirmed reports in this large scale experiment. However, this threat seems limited because all target programs are well-maintained so that these programs may not have many new bugs.

5 EXPERIMENT RESULT

For all comparison in the experiments in this section, we applied Wilcoxon test with a significance level 0.05 to show the statistical significance. All comparison results in this section are statistically significant unless mentioned otherwise. The experiment data are available at http://swtv.kaist.ac.kr/tools/conbrio.

5.1 Experiment Data

5.1.1 Data on Extended Units and Calling Contexts. For the 15 known crash bug benchmark programs, each extended unit constructed by CONBRIO contains 6.2 functions on average. CONBRIO generated 3.0 calling contexts per target function where each calling context has 6.6 functions on average. Call-graph distance techniques with bound 3, 6, and 9 generate an extended unit that contains 5.8, 13.8, and 22.5 functions on average, respectively. Also, they generate 5.9, 11.1, and 24.3 calling contexts per target function on average, respectively.

5.1.2 Data on Unit Tests Generated and Alarm Filtering. For the 15 known crash bug benchmark programs, CONBRIO spent 1.8 hours to generate 7,979,781 unit tests for 15,915 target functions and 2.3 hours for the symbolic alarm filtering using Z3 on 100 quad-core machines. Z3 reports that a symbolic calling context formula with a violating symbolic unit execution consists of 1.5 million clauses on 0.1 million Boolean variables on average and its maximum memory usage is around 7.6 GB. Call-graph distance techniques with a distance bound 0, 3, 6, and 9 spent the almost same 1.8 hours for unit test generation (i.e., most target functions reach the timeout) and 0, 2.6, 3.9, and 6.3 hours for the symbolic alarm filtering, respectively.

CONBRIO covered 69.8% to 88.0% of the branches of a target program (82.5% on average) with the unit tests and the given system test cases (i.e., the unit tests increase the branch coverage 25.2% more on average (= 82.5% - 57.3% where 57.3% is the average branch coverage achieved by the system test cases (see the last row of Table 1)).

5.2 RQ1: Bug Detection Ability

Table 3 describes a number of the target bugs detected by the concolic unit testing techniques and shows that CONBRIO has high bug detection ability. CONBRIO and static call-graph distance technique with bound zero (i.e., SUT) achieve the highest bug detection ability (i.e., 91.0% (=61/67)) (but SUT achieves this at the cost of many false alarms (see Sect. 5.3)). Note that the given system tests do not detect any of the target bugs. In addition, we applied concolic testing at system level using distributed concolic testing tool SCORE [23] with the same amount of total time on 100 machines but found that no target bug was detected.

As a distance bound of the call-graph distance techniques increases to 3, 6, and 9, the number of detected bugs severely decreases to 51, 41, and 39, respectively because larger symbolic search space should be explored within the timeout.

Among the undetected six target bugs (=67-61), three target bugs in bash, grep, and gcc were missed because concolic execution did not cover corresponding buggy statements within the timeout, two bugs in flex and h264ref were missed because of the alarm filtering strategy, and one in vim was missed because a unit execution covered the corresponding buggy statement and an assert statement but did not violate the assert statement.

5.3 RQ2: Bug Detection Precision

Table 4 describes a number of false alarms and a ratio of false alarms per true alarm of the techniques and shows that CONBRIO achieves high bug detection precision. Among the techniques, CONBRIO raises the lowest number of false alarms (5.3% in Table 1) and the lowest false alarms per true alarm ratio (i.e., 4.5 false alarms per true alarm on average).\(^6\)

\(^6\)The static alarm reduction heuristics of CONBRIO decrease the number of false alarms (23.5 to 20.5 on average) and the number of false alarms per true alarm (5.2 to
### Table 4: Numbers of false alarms and ratios of false alarms per true alarm of the concolic unit testing techniques

<table>
<thead>
<tr>
<th>Target programs</th>
<th>Static call graph distance techniques</th>
<th>CONBRO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># of F/T</td>
<td># of F/T</td>
</tr>
<tr>
<td></td>
<td>false alarm</td>
<td>false alarm</td>
</tr>
<tr>
<td>Bash-2.0</td>
<td>484 96.8</td>
<td>137 45.3</td>
</tr>
<tr>
<td></td>
<td>Flex-2.4</td>
<td>142 71.0</td>
</tr>
<tr>
<td></td>
<td>Grp-2.0</td>
<td>120 40.0</td>
</tr>
<tr>
<td></td>
<td>Gzip-1.97</td>
<td>33 16.5</td>
</tr>
<tr>
<td></td>
<td>Make-3.75</td>
<td>644 221.3</td>
</tr>
<tr>
<td></td>
<td>Sed-1.17</td>
<td>31 15.5</td>
</tr>
<tr>
<td></td>
<td>Vim-5.0</td>
<td>906 181.2</td>
</tr>
<tr>
<td></td>
<td>Perl-5.87</td>
<td>392 65.3</td>
</tr>
<tr>
<td></td>
<td>Bzip2-1.0</td>
<td>34 17.0</td>
</tr>
<tr>
<td></td>
<td>Gcc-3.2</td>
<td>2026 1447.9</td>
</tr>
<tr>
<td></td>
<td>Gnome-3.14</td>
<td>791 197.8</td>
</tr>
<tr>
<td></td>
<td>Hminer-2.042</td>
<td>162 54.0</td>
</tr>
<tr>
<td></td>
<td>Sjeng-11.2</td>
<td>108 54.0</td>
</tr>
<tr>
<td></td>
<td>Libquantum-0.24</td>
<td>55 18.3</td>
</tr>
<tr>
<td></td>
<td>Hz4/4/5</td>
<td>232 46.4</td>
</tr>
<tr>
<td>Average</td>
<td>412 82.7</td>
<td>97.6 22.9</td>
</tr>
</tbody>
</table>

The static call-graph distance technique with distance 0 (i.e., SUT) suffers the largest number of false alarms (412.0 false alarms per target program on average). CONBRO raises only 5.0% (20.5/412.0), 21.0%, and 46.4% and 61.6% of the false alarms raised by the static call-graph distance techniques with distance bounds 0, 3, 6, and 9 on average, respectively (see the last row of the table).

### 5.4 RQ3. Effectiveness of the Symbolic Alarm Filtering

The comparison of the experiment results of CONBRO and CONBRIO without the alarm filtering strategy using symbolic calling context formulas (Sect. 3.5) demonstrates that the alarm filtering strategy improves bug detection precision significantly. In other words, CONBRO without the alarm filtering strategy detects two more target bugs (i.e., 63 bugs) in all target programs but with five times higher false alarm ratio (i.e., 20.3 false alarms per true alarm on average). Although the symbolic alarm filtering spent more time (2.3 hours) than the unit test generation (1.8 hours), this strategy is worthwhile to apply to improve bug detection precision. Detailed experiment data is available at http://swtv.kaist.ac.kr/tools/conbrio.

### 5.5 RQ4. Effect of the Function Selection Strategy on Bug Detection Ability and Precision

The comparison on the experiment results of CONBRO and the call-graph distance technique confirms that the idea of including only closely relevant functions to a target function based on the proposed dependency metric in extended units and calling contexts is effective. For example, CONBRO and the call-graph distance technique with bound 3 generate an extended unit of a similar size (i.e., 6.2 vs. 5.8 functions on average) and the amount of generated calling contexts are also comparable (3.0 calling contexts each of which has 6.6 functions vs. 5.9 calling contexts each of which has 2.8 functions on average) (see Sect. 5.1.1). The time taken to generate unit test executions is almost same 1.8 hours and the time taken to apply the alarm filtering strategy is also similar (2.3 vs 2.6 hours).

However, CONBRO achieves much higher bug detection ability and precision than the call-graph distance technique with bound 3 (i.e., 91.0% vs 76.1% (51/67) for bug detection ability and 4.5 vs. 22.9 false alarms per true alarm on average). With larger distance bounds 6 and 9, a number of the detected bugs drops to 41 and 39 and the false alarm ratio decreases to 14.6 and 11.5 respectively, which is still three to two times less precise than CONBRO.

### 5.6 RQ5. Effectiveness of Detecting New Crash Bugs

CONBRO detects 14 new crash bugs in the seven target programs. CONBRO detects five new crash bugs in autotrace, two bugs in each of abc2mp3s, gzip2mpg, and mp3gain, one bug in each of bib2xml, eog, and jpegtran, and no bug in catdvi and xpdf. Note that we have confirmed the 14 new crash bugs by manually creating system-level test cases that crash a target program due to the bugs detected by CONBRO. CONBRO raises 71 false alarms over all target programs and its true to false alarm ratio for each program ranges from 1:3.0 to 1:6.0 (1:4.3 on average except catdvi and xpdf). We have reported these 14 new crash bugs to the original developers and been waiting the responses from them (detailed example and explanation of the newly detected bugs are available at http://swtv.kaist.ac.kr/tools/conbrio).

### 6 RELATED WORK

#### 6.1 Concolic Unit Testing Techniques

There exist concolic unit testing techniques (e.g., [7, 32, 37, 40]) which require a user to build symbolic unit test drivers and stubs. DART [16] generates symbolic unit test drivers (but not symbolic stubs) like SUT (Sect. 2.2) and test inputs for C programs. CONBOL [25] generates symbolic unit test drivers/stubs and test inputs targeting large-scale embedded C programs. DART and CONBOL generate symbolic unit test drivers without utilizing contexts of a target function $f$ and may suffer many false alarms. Chakrabarti and Godefroid [9] developed a unit testing technique which statically partitions a static call graph using topological information and tests each partition as a unit through symbolic execution. This technique may suffer many false alarms because the obtained partitions may not represent groups of relevant functions due to insufficient information to generate partitions (i.e., using only topological information of a static call graph without semantic or dynamic information). Their tool is not publicly available and the paper does not report bug detection ability nor precision [9]. Tomb. et al [41] reported that interprocedural program analysis with deeper call depth bound raise fewer false alarms. However, they did not report how to set a proper call depth bound.

Recently, UC-KLEE [34] directly starts symbolic execution from a target function using lazy initialization [21]. Through the manual

---

5 CONBRO generated 3.3 calling contexts per target function each of which has 4.3 functions on average. It spent 11.2 minutes to generate 725,584 unit tests for 4.040 target functions and 20.7 minutes to apply the symbolic alarm filtering on 100 machines and covered 80.9% of the branches on average.

6 DART [16] bypasses the false alarm issue by targeting public API functions of libraries which should work with all possible inputs.
analysis of the thousands of alarms, the authors of UC-KLEE detected 67 new bugs in BIND, openssl1, Linux kernel and its true to false alarm ratio is 1:5.7 on average. We could not directly compare CONBRIO with UC-KLEE because UC-KLEE is not publicly available and BIND, openssl1, and Linux kernel (million lines of code) are too large to manually analyze alarms.

6.2 Random Method Sequences Generation Techniques for Object-Oriented Programs

Randoop [29] invokes a random sequence of public methods including constructors of a target method’s class. MSeqGen [39] mines code bases to extract relevant method sequences of a target class under test and extends such method sequences with symbolic execution for high coverage. EvoSuite [13, 14] tests Java methods using search-based strategies with symbolic execution. TestFul [5] combines genetic algorithm and a local search to improve the speed of Java unit test generation. Garg et al. [15] improves Randoop by generating input test cases of the generated method sequence using concolic testing for C++ programs. These techniques may also suffer false alarms due to infeasible test inputs/method sequences generated. For example, Gross et al. [19] reported that Randoop raised 181 alarms without detecting any bug (i.e., all alarms were false ones) on five Java programs although the authors of Randoop reported that Randoop’s true to false alarm ratio is 1:0.67 on 8 Java libraries and 6.NET libraries on average [29]. Fraser et al. [14] reported that the statistically estimated true to false alarm ratios range from 1:0.6 to 1:4.2 in their experiments on randomly selected 100 projects hosted on sourceforge.net. Garg et al. [15] does not report detected bugs or false alarm ratios but branch coverage obtained using the proposed technique on eight programs (except grueschon on which the authors reported nine new bugs and that a true to false alarm ratio was 1:1.0). In spite of the lack of explicit context information (e.g., class/object information) in C programs, CONBRIO detects bugs precisely in C programs (i.e., a true to false alarm ratio is 1:4.5 on average) while keeping high bug detection ability (i.e., 91.0% of the target bug detected on average).

The aforementioned papers report only bug detection precision (RQ2), not bug detection ability (RQ1), which makes fair comparison of these techniques is that the executions of the generated unit tests just replay the same behaviors [12, 31] (or similar behaviors [20]) of a target unit in already performed system testing (i.e., they are applicable to only regression testing of evolving software, not to a single version of software). Also, the aforementioned papers do not report bug detection ability nor precision.

7 CONCLUSION AND FUTURE WORK

We have presented an automated concolic unit testing technique CONBRIO which generates extended units to closely mimic the real contexts of a target function $f$ and filters out false alarms using symbolic calling context formulas of $f$ using relevant functions to $f$. Through the experiments, CONBRIO demonstrates both high bug detection ability (91.0% of all target bugs detected) and high bug detection precision (a true to false alarm ratio is 1:4.5). Furthermore, CONBRIO detects 14 new crash bugs in the latest versions of the nine target C programs studied in other papers on crash bug detection techniques.

As future work, to improve the precision of automated unit testing further, we plan to refine the function dependency metric by analyzing more semantic characteristic of target program executions. Also, we will improve bug detection precision by synthesizing a common-likely symbolic calling context based on multiple calling contexts of a target function $f$.

ACKNOWLEDGEMENT

This work is supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (No. NRF-2017M3C4A7068175 and No. NRF-2017M3C4A7068177), and Basic Science Research Program through NRF funded by MSIT (NRF-2016R1A2B4008113), and Basic Science Research Program through NRF funded by the Ministry of Education (NRF-2017R1D1A1B03035851).

REFERENCES
