# 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.

## CCS CONCEPTS

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

**ACM Reference Format:**
Yunho Kim, Yunja Choi, and Moonzoo Kim. 2018. Precise Concolic Unit Testing of C Programs using Extended Units and Symbolic Alarm Filtering . In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180253

## 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). 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 (**CON**colic unit testing with sym-**B**olic ala**R**m f**I**ltering using symbolic calling c**O**ntexts) 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]

We have applied CONBRIO to 15 real-world C programs in SIR [11] and SPEC2006 [38] benchmarks and CONBRIO shows both


*ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180253

[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).

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, CONBRIO 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:

- CONBRIO 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 CONBRIO 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 CONBRIO, 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 CONBRIO. Section 4 explains the experiment setup to evaluate CONBRIO 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 $drv_i^f$s each of which generates argument values $a_1^i, ...a_n^i$, global variable values $v_1^i, ...v_m^i$ and finally invokes $f$ with these input values. Also, a tester builds stub functions $s_{g_1}^i...s_{g_l}^i$ 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: [2]

- *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 `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 `array` through an index `x`) and `h(x)` (line 15). A concolic unit testing technique generates a unit test driver `driver_f` and symbolic stubs `stub_g` and `stub_h` for `f`. Also, it modifies `f` to call `stub_g` and `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`. `driver_f` invokes `f` with a symbolic argument `arg1` (lines 2–3) where `int arg1 = SYM_int()` sets `arg1` as a symbolic integer value (line 2). `stub_g`

[2] This subsection is excerpted from [25].

```
01:// x and y are inputs of a target program
02:int main(int x,int y){
03:  return (x>0) ? a1(x,y) : a2(y);}
04:
05:int a1(int x, int y){if(y>0) return b(x); else return 0;}
06:int a2(int x){if(x>0) return b(x);}
07:int b(int x){if(x>0) return f(x);else return 0;}
08:
09:// Target function under test
10:int f(int x){
11:  int array[5] = {1,3,5,7,9}, result;
12:  if (g(x) != 0){ //=> if (stub_g(x) != 0) {
13:    // => assert(0<=x && x<5);
14:    result = array[x];
15:  }else result=h(x);//=> else result=stub_h(x);
16:  return result;}
17:
18:int g(int x){ return (x<5)? 1:0;}
19:
20:int h(int x){ return x + 2;}
```

**Figure 1: Target program with a target function** `f`

```
01: int driver_f(){
02:   int arg1 = SYM_int();
03:   f(arg1);}
04:
05: int stub_g(int x){}
06:   int ret = SYM_int();
07:   return ret;}
08:
09: int stub_h(int x){
10:   int ret = SYM_int();
11:   return ret;}
```

**Figure 2: Generated unit test driver and stubs for** `f`

and `stub_h` return symbolic integer values as `g` and `h` return integer values (lines 5–7 and lines 9–11 respectively). Concolic execution of `driver_f` violates the OOB assertion at line 13 of `f` if a unit test execution satisfies the following two conditions:

- a symbolic argument `arg1` to `f` (line 3 of `driver_f`) is larger than or equal to the size of `array` (e.g. `arg1` is 5)
- `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 `arg1`≥ 5 (line 18 of Figure 1) unlike `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 `main`, `a1`, `a2`, `b`, `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 test executions (Section 3.1).
2. It checks function relevance by calculating *dependency* of a target function $f$ on other function $g$ using *conditional probability* $p(g|f)$ based on the observed function call profiles (Section 3.2). 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 $a^i$s. We use $ctx(f,k)$ to indicate $k$th calling context of $f$.

   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 $\sigma_{f v_i}$ that represents executions violating a given assertion $v_i$ in $f$ (Section 3.4).
5. It filters out an alarm raised at $v_i$ by checking the feasibility of $\sigma_{f v_i}$ with regard to $f$'s *calling contexts* (see Step 3). For this purpose, CONBRIO constructs $f$'s *symbolic calling context formulas* $\Sigma_{ctx(f,k)}$ and uses a SMT solver to check satisfiability of $\sigma_{f v_i}$ (see Step 4) conjuncted with $\Sigma_{ctx(f,k)}$ (Section 3.5).
   If the result is UNSAT for all calling contexts (i.e., there exists *no* feasible execution in any calling context of $f$ to make $\sigma_{f v_i}$ feasible), a target alarm is considered as false and ignored. Otherwise (i.e., the result is SAT with at least one calling context), a corresponding alarm is reported as a violation of $v_i$ in $f$.

### 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 `main(x,y)`: (-1,1), (1,1), and (5,1). Then, the function call profiles are obtained as follows: {`main`→`a2`, `a2`→`b`, `b`→`f`,`f`→`g`} with (-1,1), {`main`→`a1`, `a1`→`b`, `b`→`f`,`f`→`g`} with (1,1), and {`main`→`a1`, `a1`→`b`, `b`→`f`,`f`→`g`, `f`→`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 $p(g|f)$. Given a static call graph $G(V,E)$ (see Def. 1) and system test executions, we compute $p(g|f)$ as follows:

- Case 1: for $g$ which is a *predecessor* of $f$ in $G(V,E)$, $p(g|f)$ is calculated as $\frac{n_1}{n_f}$ where $n_1$ is a number of system executions where $g$ calls $f$ directly or transitively.
- Case 2: for $g$ which is a *successor* of $f$ in $G(V,E)$, $p(g|f)$ is calculated as $\frac{n_2}{n_f}$ where $n_2$ is a number of system executions where $f$ calls $g$ directly or transitively.
- Case 3: for $g$ which is a *successor* and *predecessor* of $f$ in $G(V,E)$ (i.e., there exists a recursive call cycle between $f$ and $g$), $p(g|f)$ is calculated as $\frac{n_3}{n_f}$ where $n_3$ is a number of system executions where $f$ calls $g$ or $g$ calls $f$ directly or transitively.

For example, Step 1 of Fig. 3 shows three test cases (-1,1), (1,1), and (5,1) and their corresponding function call profiles for the program

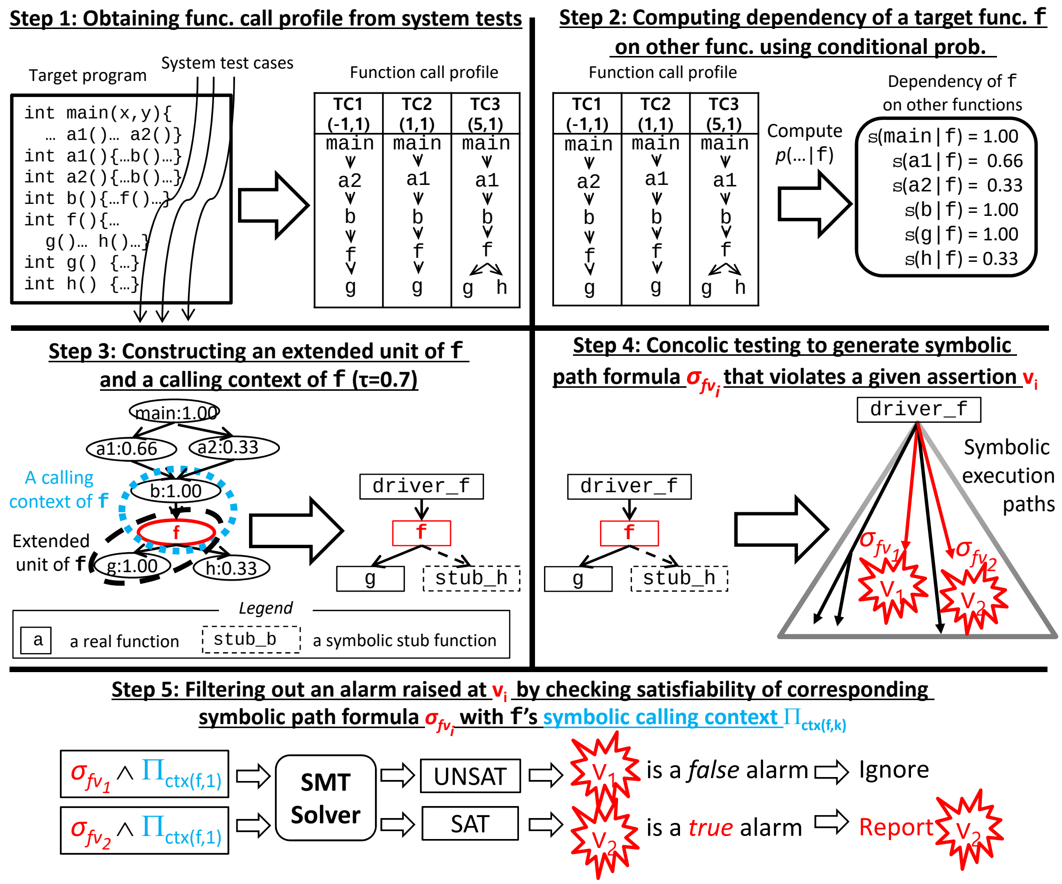


**Figure 3: Overall process of CONBRIO**

in Fig. 1. Based on the profiles, we calculated dependency of $f$ on other functions as follows:

- $p(\mathtt{main}|\mathtt{f}) = 1.00\ (= \frac{n_1}{n_f} = \frac{3}{3})$
- $p(\mathtt{a1}|\mathtt{f}) = 0.66\ (= \frac{n_1}{n_f} = \frac{2}{3})$
- $p(\mathtt{a2}|\mathtt{f}) = 0.33\ (= \frac{n_1}{n_f} = \frac{1}{3})$
- $p(\mathtt{b}|\mathtt{f}) = 1.00\ (= \frac{n_1}{n_f} = \frac{3}{3})$
- $p(\mathtt{g}|\mathtt{f}) = 1.00\ (= \frac{n_2}{n_f} = \frac{3}{3})$
- $p(\mathtt{h}|\mathtt{f}) = 0.33\ (= \frac{n_2}{n_f} = \frac{1}{3})$

## 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 $p$ as a *predecessor* of $f$ if there exists a path from $p$ to $f$. We call a node $s$ as 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., $p(\mathtt{g}|\mathtt{f}) \geq \tau$ but $p(\mathtt{h}|\mathtt{f}) < \tau$ where $\tau = 0.7$). Finally, `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$\longrightarrow$ f since f has high dependency on b, but not a1 nor a2 (i.e., $p(\mathtt{b}|\mathtt{f}) \geq \tau$ but $p(\mathtt{a1}|\mathtt{f}), p(\mathtt{a2}|\mathtt{f}) < \tau$).

### 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 labeled 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).

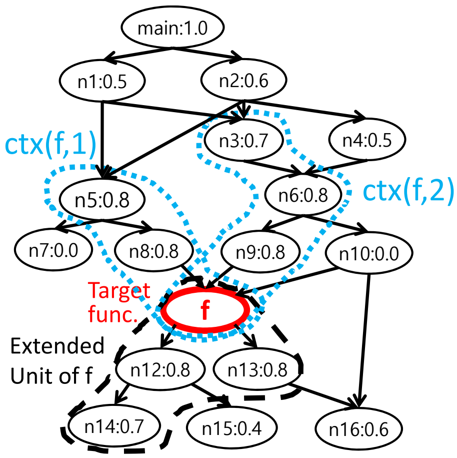


**Figure 4: Static call graph showing an extended unit and two calling contexts of $f$ ($ctx(f,1)$ and $ctx(f,2)$) with $\tau = 0.7$**

In addition, as a false alarm reduction heuristic, CONBRIO adds `SYM_assume(`*expr*`)` [3] 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.

*3.3.2 Constructing Calling Contexts.* In a static call graph $G(V,E)$ labelled with dependency of $f$, we define a *calling context* of $f$ as a maximal call path from a predecessor node of $f$ to $f$ as follows.

Definition 2. An $i$th *calling context* of $f \in V$ (saying $ctx(f,i)$) is a maximal call path $a^1 \rightarrow a^2 \rightarrow \ldots \rightarrow f$ in a static call graph $G(V,E)$ satisfying the following conditions:

- $a^1$ is a predecessor of $f$
- for all $a^j$ in $ctx(f,i)$, $p(a^j|f) \geq \tau$
- 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, $ctx(f,1) = n5 \rightarrow n8 \rightarrow f$. Similarly, $ctx(f,2) = n3 \rightarrow n6 \rightarrow n9 \rightarrow f$.

## 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 $\sigma_{f(v_i,j)}$ 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 $\sigma_{fv_i}$ to denote $\bigvee_j \ \sigma_{f(v_i,j)}$.

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 $\sigma_{fv_i}$ with $f$'s Symbolic Calling Context Formula

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

- For each function $a^j$ in a calling context of $f$ (i.e. $ctx(f,k)$), CONBRIO obtains $SE_{a^j}$ which is a set of symbolic execution path formulas of $a^j$.
  Note that this task is the same task in Step 4 (Sect. 3.4). If $a^j$ was already tested as a target function and $SE_{a^j}$ was generated in Step 4, this alarm filtering step (Step 5) reuses $SE_{a^j}$.
- CONBRIO obtains a slice of $SE_{a^j}$ with regard to $a^{j+1}$ (saying $Slice(SE_{a^j}, a^{j+1})$) as follows:
  $Slice(SE_{a^j}, a^{j+1}) \stackrel{def}{=} \{\sigma' | \sigma'$ is a prefix of $\sigma \in SE_{a^j}$ such that $\sigma$ contains invocation of $a^{j+1}$ and $\sigma'$ does not contain a suffix of $\sigma$ after the invocation$\}$.
  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 $\sigma_{x(y,z)}$ is $z$th 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 $a^1$ (i.e., $Slice(SE_{a^1}, a^2)$), $a^2$ (i.e. $Slice(SE_{a^2}, a^3)$), ... of $ctx(f,k)$ until reaching $f$ using logical conjunction. Thus, $\Pi_{ctx(f,k)}$ with $ctx(f,k) = a^1 \rightarrow a^2 \rightarrow \ldots \rightarrow f$ is defined as follows:
  $$\Pi_{ctx(f,k)} \stackrel{def}{=} \bigwedge_{a^j \in (ctx(f,k)-\{f\})} (\bigvee_{\sigma_l \in Slice(SE_{a^j}, a^{j+1})} \sigma_l)$$
  For example, Fig. 5 shows $\Pi_{ctx(f,1)}$ with $ctx(f,1) = n5 \rightarrow n8 \rightarrow f$ in Fig. 4 as follows (see thick blue arrows representing $\sigma_{n5(n8,1)}$, $\sigma_{n5(n8,2)}$, $\sigma_{n8(f,1)}$ and $\sigma_{n8(f,2)}$):
  $$\Pi_{ctx(f,1)} = \Pi_{n5 \rightarrow n8 \rightarrow f} = (\sigma_{n5(n8,1)} \vee \sigma_{n5(n8,2)}) \wedge (\sigma_{n8(f,1)} \vee \sigma_{n8(f,2)})$$

Finally, CONBRIO applies a SMT solver to $\sigma_{fv_i} \wedge \Pi_{ctx(f,k)}$ 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 $\sigma_{fv_i}$ feasible), a target alarm is considered as false and ignored. Otherwise (i.e. , a result is SAT with at least one

[3] `SYM_assume(`*expr*`)` is a macro of `if(!expr) exit(0);`.

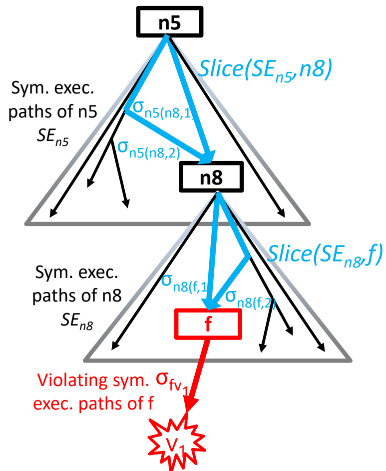


**Figure 5: Violating symbolic path formula $\sigma_{f v_i}$ and a symbolic calling context of $f$ (i.e., $\Pi_{ctx(f,1)}$) with $ctx(f,1)$ in Fig. 4**

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?

**Table 1: Target programs and bugs for RQ1 to RQ4**

| Target programs and versions | Lines | # of func. | # of sys. test cases | Branch cov. (%) | Func cov. (%) | # of target bugs |
|---|---|---|---|---|---|---|
| Bash-2.0 | 32714 | 1214 | 1100 | 46.2 | 89.0 | 6 |
| Flex-2.4.3 | 7471 | 147 | 567 | 45.7 | 93.9 | 2 |
| Grep-2.0 | 5956 | 132 | 809 | 50.3 | 94.7 | 5 |
| Gzip-1.0.7 | 3054 | 82 | 214 | 55.8 | 87.8 | 2 |
| Make-3.75 | 28715 | 555 | 1043 | 64.5 | 87.9 | 3 |
| Sed-1.17 | 4085 | 73 | 360 | 47.3 | 87.7 | 2 |
| Vim-5.0 | 66209 | 1749 | 975 | 35.8 | 91.0 | 6 |
| Perl-5.8.7 | 79873 | 2240 | 1201 | 52.3 | 95.0 | 6 |
| Bzip2-1.0.3 | 4737 | 114 | 6 | 67.4 | 93.9 | 2 |
| Gcc-3.2 | 342561 | 5553 | 9 | 43.7 | 96.2 | 15 |
| Gobmk-3.3.14 | 154583 | 2682 | 1354 | 65.2 | 92.0 | 5 |
| Hmmer-2.0.42 | 35992 | 539 | 4 | 75.6 | 94.1 | 3 |
| Sjeng-11.2 | 10146 | 144 | 3 | 77.9 | 91.7 | 2 |
| Libquantum-0.2.4 | 2255 | 101 | 3 | 68.5 | 93.1 | 3 |
| H264ref-9.3 | 51578 | 590 | 6 | 63.6 | 88.0 | 5 |
| Sum | 829929 | 15915 | 7654 | N/A | N/A | 67 |
| Average | 55328.6 | 1061.0 | 510.3 | 57.3 | 91.7 | 4.5 |

**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 2: Target programs for RQ5**

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

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).

*4.2.2 Unknown Crash Bug Benchmark.* The unknown crash bug benchmark programs were selected from the literature on crash bug detection techniques. This is because SIR and SPEC benchmark programs do not satisfy the following criteria: we selected target programs whose sizes are 1,000 to 100,000 LoC and which have more than three crash bug fixes in the last three years (i.e. between April 2014 to April 2017). We excluded very large programs due to huge manual effort required to check validity of alarms. We also excluded programs with three or less crash bug fixes in the last three years because such programs may not have a crash bug.

To obtain new crash bug benchmark programs, we surveyed papers on crash bug detection techniques published in major SE (ICSE, FSE, ASE, ISSTA), PL (PLDI, POPL, SPLASH), and security conferences (IEEE S&P, ACM CCS, USENIX Security) in the last three years and obtained the nine relevant papers [3, 8, 18, 27, 34, 35, 43–45]. Then, we applied the above criteria to the latest versions of the target programs studied in these papers and obtained the nine target programs in Table 2. Again, for all target programs, we used all system test cases provided in the target program versions and no system test case violated the crash assertions.

## 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). [4] These unit testing techniques have been implemented in 1,000 lines of C++ code using CROWN [1].

## 4.4 Measurement

We consider that 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 $s$ 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 $\sigma_{fv}$ that covers a buggy statement and violates an assert statement in a target function $f$.
- We can confirm that $\sigma_{fv}$ is feasible at system level by manually creating a system-level test that includes $\sigma_{fv}$ and violates the assert statement (we compared execution traces of $\sigma_{fv}$ 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. [5] 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] We report the experiment using only DFS and modified DFS since the experiments using other search strategies such as random negation and CFG heuristic show only negligible difference.

[5] We selected timeout as 180 seconds because exploratory study with timeout 60, 180, 300, and 600 seconds suggested that timeout beyond 180 seconds had negligible effect on the overall experiment results of CONBRIO and the other techniques.

## 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 Wilcoxn 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%p 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)).

**Table 3: Numbers of the target bugs detected by the concolic unit testing techniques**

| Target program | # of target bugs | Bound of static call graph distance tech. | | | | CONBRIO |
|---|---|---|---|---|---|---|
| | | 0(SUT) | 3 | 6 | 9 | |
| Bash-2.0 | 6 | 5 | 3 | 3 | 3 | 5 |
| Flex-2.4.3 | 2 | 2 | 1 | 1 | 1 | 1 |
| Grep-2.0 | 5 | 3 | 4 | 2 | 2 | 4 |
| Gzip-1.0.7 | 2 | 2 | 1 | 1 | 1 | 2 |
| Make-3.75 | 3 | 3 | 3 | 2 | 2 | 3 |
| Sed-1.17 | 2 | 2 | 2 | 2 | 2 | 2 |
| Vim-5.0 | 6 | 5 | 4 | 2 | 2 | 5 |
| Perl-5.8.7 | 6 | 6 | 5 | 4 | 3 | 6 |
| Bzip2-1.0.3 | 2 | 2 | 2 | 2 | 2 | 2 |
| Gcc-3.2 | 15 | 14 | 12 | 9 | 8 | 14 |
| Gobmk-3.3.14 | 5 | 4 | 3 | 3 | 3 | 5 |
| Hmmer-2.0.42 | 3 | 3 | 3 | 3 | 3 | 3 |
| Sjeng-11.2 | 2 | 2 | 2 | 2 | 2 | 2 |
| Libquantum-0.2.4 | 3 | 3 | 2 | 2 | 2 | 3 |
| H264ref-9.3 | 5 | 5 | 4 | 3 | 3 | 4 |
| Sum | 67 | 61 | 51 | 41 | 39 | 61 |

## 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 (i.e., 20.5 false alarms per target program on average) and the lowest false alarms per true alarms 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**

| Target programs | Static call-graph distance techniques | | | | | | | | CONBRIO | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 (SUT) | | 3 | | 6 | | 9 | | | |
| | # of false alarms | F/T alarm ratio | # of false alarms | F/T alarm ratio | # of false alarms | F/T alarm ratio | # of false alarms | F/T alarm ratio | # of false alarms | F/T alarm ratio |
| Bash-2.0 | 484 | 96.8 | 137 | 45.7 | 69 | 23.0 | 54 | 18.0 | 18 | 3.6 |
| Flex-2.4.3 | 142 | 71.0 | 25 | 25.0 | 12 | 12.0 | 12 | 12.0 | 6 | 6.0 |
| Grep-2.0 | 120 | 40.0 | 34 | 8.5 | 18 | 9.0 | 18 | 9.0 | 13 | 3.3 |
| Gzip-1.0.7 | 33 | 16.5 | 7 | 7.0 | 3 | 3.0 | 3 | 3.0 | 5 | 2.5 |
| Make-3.75 | 664 | 221.3 | 106 | 35.3 | 59 | 29.5 | 46 | 23.0 | 9 | 3.0 |
| Sed-1.17 | 31 | 15.5 | 9 | 4.5 | 4 | 2.0 | 4 | 2.0 | 5 | 2.5 |
| Vim-5.0 | 906 | 181.2 | 207 | 51.8 | 123 | 61.5 | 72 | 36.0 | 25 | 5.0 |
| Perl-5.8.7 | 392 | 65.3 | 187 | 37.4 | 64 | 16.0 | 44 | 14.7 | 57 | 9.5 |
| Bzip2-1.0.3 | 34 | 17.0 | 12 | 6.0 | 7 | 3.5 | 7 | 3.5 | 10 | 5.0 |
| Gcc-3.2 | 2026 | 144.7 | 503 | 41.9 | 195 | 21.7 | 147 | 18.4 | 79 | 5.6 |
| Gobmk-3.3.14 | 791 | 197.8 | 133 | 44.3 | 62 | 20.7 | 45 | 15.0 | 39 | 7.8 |
| Hmmer-2.0.42 | 162 | 54.0 | 48 | 16.0 | 22 | 7.3 | 22 | 7.3 | 12 | 4.0 |
| Sjeng-11.2 | 108 | 54.0 | 13 | 6.5 | 7 | 3.5 | 7 | 3.5 | 8 | 4.0 |
| Libquantum-0.2.4 | 55 | 18.3 | 9 | 4.5 | 4 | 2.0 | 4 | 2.0 | 5 | 1.7 |
| H264ref-9.3 | 232 | 46.4 | 34 | 8.5 | 15 | 5.0 | 15 | 5.0 | 17 | 4.3 |
| Average | 412.0 | 82.7 | 97.6 | 22.9 | 44.3 | 14.6 | 33.3 | 11.5 | 20.5 | 4.5 |

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). CONBRIO 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 CONBRIO 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, CONBRIO 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 CONBRIO and the call-graph distance techniques 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, CONBRIO 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, CONBRIO 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 CONBRIO.

## 5.6 RQ5. Effectiveness of Detecting New Crash Bugs

CONBRIO detects 14 new crash bugs in the seven target programs. CONBRIO detects five new crash bugs in `autotrace`, two bugs in each of `abcm2ps`, `gif2png`, and `mp3gain`, one bug in each of `bib2xml`, `eog`, and `jpegtran`, and no bug in `catdvi` and `xpdf`. [7]

Note that we confirmed the 14 new crash bugs by manually creating system-level test cases that crash a target program due to the bugs detected by CONBRIO. CONBRIO raises 71 false alarms over the 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. [8]

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

4.5 on average) without decreasing the bug detection ability (i.e. CONBRIO without the static alarm reduction heuristics detects the same 61 bugs and raises 23.5 false alarms per target program on average).

[7] CONBRIO 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.

[8] 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`, `openssl`, 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`, `openssl`, 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 `gnuchess` 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 between these techniques and CONBRIO difficult. This is because these techniques may improve a true to false alarm ratio at the cost of missing bugs. Because of such trade-off between precision and recall of bug detection, we studied and reported both bug detection ability and precision.

## 6.3 Automated Unit Testing Techniques based on System Tests

Elbaum et al. [12] proposed a technique to generate unit tests from system tests; the technique captures program states before and after an invocation of a target function $f$ to generate unit test inputs and oracles for $f$. OCAT [20] captures object instances during system executions and generates unit tests using Randoop with the captured object and the mutated object instances as seed objects. GenUTest [31] automatically generates unit tests and mock objects using captured method sequences during system testing. A limitation 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

[1] [n. d.]. CROWN: Concolic testing for Real-wOrld softWare aNalysis. ([n. d.]). http://github.com/swtv-kaist/CROWN.

[2] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. 2008. Finding Bugs in Dynamic Web Applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 261–272.

[3] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. 2014. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1083–1094.

[4] Radu Banabic, George Candea, and Rachid Guerraoui. 2014. Finding Trojan Message Vulnerabilities in Distributed Systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 113–126.

[5] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. 2010. TestFul: An Evolutionary Test Approach for Java. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society, Washington, DC, USA, 185–194.

[6] Ella Bounimova, Patrice Godefroid, and David Molnar. 2013. Billions and Billions of Constraints: Whitebox Fuzz Testing in Production. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 122–131.

[7] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 209–224.

[8] S. K. Cha, M. Woo, and D. Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy*. 725–741. https://doi.org/10.1109/SP.2015.50

[9] 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.

[10] Christoph Csallner and Yannis Smaragdakis. 2004. JCrasher: An Automatic Robustness Tester for Java. *Software Practical Experience* 34, 11 (Sept. 2004), 1025–1050.

[11] 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.

[12] S. Elbaum, H. Chin, M. Dwyer, and M. Jorde. 2009. Carving and Replaying Differential Unit Test Cases from System Test Cases. *IEEE Transactions on Software Engineering (TSE)* 35, 1 (Jan 2009), 29–45.

[13] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419.

[14] Gordon Fraser and Andrea Arcuri. 2015. 1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite. *Empirical Software Engineering* 20, 3 (June 2015), 611–639.

[15] Pranav Garg, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. 2013. Feedback-directed Unit Test Generation for C/C++ Using Concolic Execution. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 132–141.

[16] 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.

[17] Patrice Godefroid, Michael Y Levin, and David A Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the 2008 Network and Distributed System Symposium*, Vol. 8. 151–166.

[18] Denis Gopan, Evan Driscoll, Ducson Nguyen, Dimitri Naydich, Alexey Loginov, and David Melski. 2015. Data-delineation in Software Binaries and Its Application to Buffer-overrun Discovery. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 145–155. http://dl.acm.org/citation.cfm?id=2818754.2818775

[19] F. Gross, G. Fraser, and A. Zeller. 2012. Search-based System Testing: High Coverage, No False Alarms. In *International Symposium on Software Testing and Analysis (ISSTA)*.

[20] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. 2010. OCAT: Object Capture-based Automated Testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. ACM, New York, NY, USA, 159–170.

[21] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. 2003. Generalized Symbolic Execution for Model Checking and Testing. In *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*. Springer-Verlag, Berlin, Heidelberg, 553–568.

[22] Moonzoo Kim, Yunho Kim, and Yoonkyu Jang. 2012. Industrial Application of Concolic Testing on Embedded Software: Case Studies. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, Washington, DC, USA, 390–399.

[23] M. Kim, Y. Kim, and G. Rothermel. 2012. A Scalable Distributed Concolic Testing Approach: An Empirical Evaluation. In *International Conference on Software Testing, Verification and Validation (ICST)*.

[24] Yunho Kim, Moonzoo Kim, YoungJoo Kim, and Yoonkyu Jang. 2012. Industrial Application of Concolic Testing Approach: A Case Study on Libexif by Using CREST-BV and KLEE. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 1143–1152.

[25] Y. Kim, Y. Kim, T. Kim, G. Lee, Y. Jang, and M. Kim. 2013. Automated unit testing of large industrial embedded software using concolic testing. In *Proceedings of the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering*. 519–528.

[26] 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–.

[27] Wei Le and Shannon D. Pattison. 2014. Patch Verification via Multiversion Interprocedural Control Flow Graphs. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 1047–1058. https://doi.org/10.1145/2568225.2568304

[28] David Molnar, Xue Cong Li, and David A. Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs. In *Proceedings of the 18th Conference on USENIX Security Symposium (SSYM'09)*. USENIX Association, Berkeley, CA, USA, 67–82.

[29] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, Washington, DC, USA, 75–84.

[30] Yongbae Park, Shin Hong, Moonzoo Kim, Dongju Lee, and Junhee Cho. 2015. Systematic Testing of Reactive Software with Non-deterministic Events: A Case Study on LG Electric Oven. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2 (ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 29–38.

[31] B. Pasternak, S. Tyszberowicz, and A. Yehudai. 2009. GenUTest: A Unit TEst and Mock Aspect Generation Tool. *Software Tools for Technology Transfer* 11, 4 (2009), 273–290.

[32] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 15–26.

[33] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. 2013. A Fast and Low-overhead Technique to Secure Programs Against Integer Overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–11.

[34] David A. Ramos and Dawson Engler. 2015. Under-constrained Symbolic Execution: Correctness Checking for Real Code. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC'15)*. USENIX Association, Berkeley, CA, USA, 49–64.

[35] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[36] Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks Before Deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '10)*. ACM, New York, NY, USA, 186–196.

[37] 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.

[38] spec2006 [n. d.]. The SPEC CPU 2006 Benchmark Suite. ([n. d.]). https://www.spec.org/cpu2006/.

[39] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: object-oriented unit-test generation via mining source code. In *European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*.

[40] 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.

[41] 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

[42] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '04)*. ACM, New York, NY, USA, 97–107.

[43] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder. 2015. High System-Code Security with Low Overhead. In *2015 IEEE Symposium on Security and Privacy*. 866–879. https://doi.org/10.1109/SP.2015.58

[44] Xiaofei Xie, Yang Liu, Wei Le, Xiaohong Li, and Hongxu Chen. 2015. S-looper: Automatic Summarization for Multipath String Loops. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 188–198. https://doi.org/10.1145/2771783.2771815

[45] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. 2015. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*. 797–812. https://doi.org/10.1109/SP.2015.54