Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction

XIANG GAO, National University of Singapore, Singapore
BO WANG[*][†], Peking University, China
GREGORY J. DUCK, National University of Singapore, Singapore
RUYI JI[†], Peking University, China
YINGFEI XIONG, Peking University, China
ABHIK ROYCHOUDHURY, National University of Singapore, Singapore

Automated program repair is an emerging technology which seeks to automatically rectify program errors and vulnerabilities. Repair techniques are driven by a correctness criterion which is often in the form of a test-suite. Such test-based repair produces over-fitting patches, where the patches produced may fail on tests outside the test-suite driving the repair. In this work, we present a repair method which fixes program vulnerabilities without the need for a voluminous test-suite. Given a vulnerability as evidenced by an exploit, the technique extracts a general constraint representing the vulnerability from sanitizers. The extracted constraint serves as a proof obligation which our synthesized patch should satisfy. The proof obligation is met by propagating the extracted constraint to locations which are deemed to be "suitable" fix locations. An implementation of our approach (ExtractFix) on top of the KLEE symbolic execution engine shows its efficacy in fixing a wide range of vulnerabilities on subjects taken from CVEs and Google's Open-source-systems OSS Fuzz framework. Ours is the first work to propose analysis based fix localization for repair. We believe that our work presents a way forward for the over-fitting problem in program repair, by generalizing observable hazards/vulnerabilities (as constraint) from a single failing test or exploit.

ACM Reference Format:
Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2019. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (December 2019), 22 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automated program repair [24] is an emerging area for automated rectification of programming errors. In the most commonly studied problem formulation, the goal is to find a (minimal) change to a given buggy program P so it passes a test-suite T—i.e., *test-suite driven program repair.* As the goal is to find changes that merely passes the test-suite T, the automatically generated patch may *over-fit* the test data, meaning that the patched program P' may still fail on program inputs/tests outside of T. The problem is particularly dangerous in the case of software vulnerabilities. Namely,

[*]Corresponding Author
[†]The second and fourth author contributed to this work while visiting National University of Singapore.

Authors' addresses: Xiang Gao, National University of Singapore, Singapore, gaoxiang@comp.nus.edu.sg; Bo Wang, Peking University, China, wangbo_15@pku.edu.cn; Gregory J. Duck, National University of Singapore, Singapore, gregory@comp.nus.edu.sg; Ruyi Ji, Peking University, China, jiruyi910387714@pku.edu.cn; Yingfei Xiong, Peking University, China, xiongyf@pku.edu.cn; Abhik Roychoudhury, National University of Singapore, Singapore, abhik@comp.nus.edu.sg.

1049-331X/2019/12-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

if the correctness specification driving the repair of P is incomplete (such as a test-suite T)—the automatically generated patch may not completely fix the vulnerability meaning that the patched program is still vulnerable. It has been shown in the past that even for manually generated fixes, 9% of the fixes are incomplete or incorrect [53]. For automatically generated fixes of program vulnerabilities, we need a stronger level of assurance about the quality of patches.

Automatically generating high quality fixes is one of the key challenges in program repair research today. Low quality fixes which over-fit the given test result from weak specifications driving the repair. The fundamental reason for the existence of over-fitting patches is that the patch space is under-constrained due to the incomplete specification given by test suites [39].

To combat the issue of over-fitting in program repair, several approaches propose program repair driven by static analysis and verification techniques. However, these approaches are usually designed to fix certain bug classes, e.g. memory/resource leak [41], null pointer de-reference [50], memory de-allocation errors [25] or concurrency bugs [27]. In addition, the static-based approaches, e.g. Phoenix [1], may introduce false positive or false negative because of the fact that they need off-the-shelf static analyzer as oracles.

In this paper, we propose a general approach to combat the over-fitting problem, specifically for fixing *security vulnerabilities.* Our key insight is that information about the underlying cause of a vulnerability can be automatically extracted, and this information can then be used to guide Automated Program Repair (APR). The information could be extracted in the form of a *constraint* that all program states must satisfy at the buggy location in order to avoid repeating the vulnerability. Then, the goal of repair is to ensure the *constraint* is always satisfied under any program states.

Our workflow begins with the detection of an exploitable vulnerability in the form of a crash, and we assume that the failing test or exploit is available. After witnessing a crash in an exploit, a constraint representation of the violated condition—i.e., the *crash constraint* or its negation, the *crash-free constraint*—can then be extracted from either the program itself (e.g. user assertion failure), or API documentation, or hardware fault (e.g., null pointer dereference), or safety properties enforced by dynamic analysis tools such as *sanitizers.* Sanitizers, such as AddressSanitizer (ASAN) [38] and UndefinedBehaviourSanitizer (UBSAN) [45], typically implement one or more *security policies* (such as memory safety) using an instrumentation framework. If the program violates the security policy, the sanitizer induces a crash and the corresponding constraint can be extracted. For example, a buffer overflow can be formalized as violation of constraint:

$$access(buffer) < base(buffer) + size(buffer)$$

This constraint is extracted at run-time when the crash is witnessed, and represents the precise condition that all patched programs must satisfy in order to avoid repeating the same underlying crash. Crucially, the crash constraint is not specific to the test input (also called exploit in security terminology) which witnessed the violation! Thus, we can use the crash constraint to guide program repair. To do so, we *propagate* the extracted constraint backwards from the crash location to one or more suitable *fix locations* by calculating the *weakest precondition.* The fix locations are decided using a *fix localization* algorithm that examines the data and control dependencies with the crash location. Next, we synthesize a patch so that weakest precondition, the extension of crash-free constraint, cannot be violated, thereby guaranteeing that the patched program cannot repeat the same crash, and thus resolving the vulnerability. Our workflow also allows the program repair system to decide between single line and multi-line fixes as shown by our experiments. We instantiate the proposed approach in a prototype named ExtractFix.

The contributions of this paper can be summarized as follows.

- *Conceptual Contribution:* We alleviate the over-fitting problem in program repair [39], albeit only for security vulnerabilities, by generating patches that generalize beyond tests. Our main insight is to extract symbolic constraints from violations in an exploit trace witnessed by sanitizer. The constraint extraction from sanitizers is made possible by automatically symbolizing variables/memory relevant to the crash.

- *Technical Contribution:* Our constraint-based program repair method has several technical novelties over and above the existing works on test-based semantic repair (e.g., [33, 35, 51]), and the recent work on SENX [14]. First of all, we provide an effective constraint/dependency based fix localization instead of relying on statistical fault localization, as in almost all existing works on test-based program repair. Secondly, we only need a single exploit trace to generalize the vulnerability whereas existing program repair works usually need a test-suite. Last but not the least, unlike existing vulnerability repair works like SENX, we are able to synthesize non-trivial patches at locations far off from the crash location, since our technique is endowed with the power of scalable constraint propagation.
- *Utilitarian Contribution:* We implement our security vulnerability repair approach in a tool named ExtractFix, and we plan to make our tool available open-source for usage by the community. We evaluate ExtractFix on 30 CVEs and show that ExtractFix can generate more correct patches than start-of-the-art program repair tools based on test-suites. The generated patches can be found in https://extractfix.github.io.

2 OVERVIEW

For our purposes, a *crash* is broadly defined to be any program termination due to control flow reaching some illegal states where some conditions/properties are violated. A crash can be caused by the violation of an explicit user assertion (e.g., `assert(`C`)`), an implicit assertion enforced by hardware/operating-system (e.g., illegal memory access), or instrumented check inserted by *sanitizers* to enforce some safety properties. Typical sanitizers, such as AddressSanitizer (ASAN) [38] and UndefinedBehaviorSanitizer (UBSAN) [45], *instrument* the program with implicit assertions that enforce additional properties, such as memory safety, type safety, integer overflows protection, etc. If a sanitizer assertion is violated, the program will abort (i.e., “crash”), usually with an error message indicating the problem. The underlying cause of a crash can be automatically extracted in the form of a *crash-free-constraint* (CFC). The CFC represents *the constraint that all program states must satisfy at the crashing location in order to avoid repeating the crash.* For example, for a user assertion violation (`assert(`C`)`) the CFC is C itself, for a `NULL`-pointer de-reference on p the *CFC* is ($p{\neq}0$), and for an array bounds overflow error on $a[i]$ the CFC is ($i{<}SIZE$) where $SIZE$ is the size of array a.If the crashing program is patched so that the *CFC* is always satisfied, then the same crash cannot be repeated for any program input.

Our basic approach of using crash-free-constraints to guide program repair faces several challenges. This first challenge is the extraction of the *CFC* from an observable crash. The observable program crash is a concrete property violation when executing a failing test or exploit, while *CFC* should capture the properties *forall* possible inputs. The *CFC* is actually symbolic constraint extracted from concrete violations. The second challenge concerns *fix localization* (FL) in order to find one (or more) suitable fix location(s). Typically, existing FL approaches, e.g. spectrum-based FL [36], rely on test cases, and FL results depend on the quality of the tests. However, high-quality tests are not always available. In a very common scenario, there is only one test in the form of an exploit when security vulnerabilities are found. The third challenge is that the fix location(s) could be different from the crash location, meaning that the extracted *CFC* must be *propagated* and possibly transformed to guide patch generation at fix location(s). Conceptually, the *CFC* at the crash location is propagated to a *CFC*′ at a given fix location satisfying the following Hoare triple:

$$\{CFC'\}\ P\ \{CFC\} \qquad \text{(CFC-Propagation)}$$

Here, P represents the program statements between fix location and crash location. CFC' is the least restrictive (weakest) precondition that will guarantee the postcondition CFC [4]. Finding CFC' involves solving CFC-Propagation. For multi-line repair, the approach is generalized and propagation is applied to multiple fix locations. The final challenge is to use *synthesis* to generate candidate patches that enforce CFC' and (by extension) CFC. Conceptually, this involves rewriting the fix location statement ρ into an alternative f such that the following Hoare logic holds:

$$\{true\}\ [\ \rho \mapsto f\ ]\ \{CFC'\}\ P\ \{CFC\} \qquad \text{(CFC-Repair)}$$

Once repaired, the program can never again enter a state where CFC is violated at the crash location, thereby resolving the crash for all program inputs regardless of origin.

2.1 Workflow

Our basic workflow consists of several components/steps, including:

(1) **Constraint Extraction**. Given a program and a single input that exercises the crash, the first step is to extract the "crash-free constraint" (*CFC*). *CFC* is extracted according to predefined templates which formulate the underlying cause of the defect.
(2) **Fix Localization**. Once the *CFC* is generated, one (or more) candidate *fix locations* are generated using a *dependency-based fix localization* algorithm. Unlike the widely used spectrum-based fault localization (SBFL) [36], our workflow use the crash location as a starting point and find candidate fix locations using control/data dependency analysis.
(3) **Constraint Propagation**. The *CFC* is a constraint over the program state at the crash location. We must *propagate* the *CFC* to an equivalent constraint CFC' at the fix location, conceptually by solving the Hoare triple (CFC-Propagation).
(4) **Patch Synthesis**. Once the fix location and propagated CFC' have been decided, the next step is to generate patch candidates. The generated patch is guaranteed to ensure that CFC' is satisfied, meaning that the *CFC* condition at the crash location can not be violated in the patched program.

Workflow Example. To illustrate our workflow, we consider the buggy from `Coreutils`. The buggy code snippet is shown in Figure 1a. Here, the snippet attempts to fill a buffer `r` with a pattern determined by variable *bits* using repeated calls to `memcpy`. The length of each `memcpy` operation is doubled inside the *for*-loop, and the final `memcpy` handles any remaining unfilled space in the buffer. Unfortunately, this contains a bug [1]. For some input (e.g., `size`=13, i.e., "unlucky thirteen"), the source and destination regions for the final `memcpy` will overlap—undefined behaviour under the `memcpy` specification. This bug may cause program crash on some platforms. Specifically, when `size`=13, the *for*-loop will terminate in the second iteration with i=6 and $size/2$=6 (integer division). Then, at line 7, the source and destination of memcpy overlap because r+(13−6)>r+6. Using an appropriate sanitizer (UBSAN), this program will crash on the final `memcpy` call.

Figure 1b shows the overall workflow of our approach. We start with the single crashing input (`size`=13) that triggers the crash on line 7 (highlighted). Step (1) generates the *CFC* corresponding to the crash according to predefined template. The *CFC* template (shown in Section 4.1) of `memcpy(p, q, s)` is defined as `p+s` ≤ `q` ∨ `q+s` ≤ `p`. In this case, CFC is

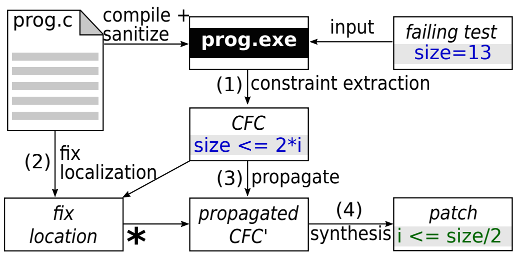
$$\texttt{(r+i+size-i}\le\texttt{r} \lor \texttt{r+size-i}\le\texttt{r+i)} \equiv \texttt{(size} \le \texttt{0} \lor \texttt{size}\le\texttt{2*i)}$$

Since size is an unsigned integer (size_t) value, we only focus on the second clause `size≤2*i` in this example. Step (2) determines candidate fix locations. One promising fix location is the *for*-condition on line 4 (highlighted) since there exists a control dependency with an assignment (`i *= 2`, line 4)

[1]https://debbugs.gnu.org/cgi/bugreport.cgi?bug=26545

```
void fillp (char *r, size_t size){
  ...
  r[2] = bits & 255;
  for (i = 3; i < size / 2 ; i *= 2)
    memcpy(r + i, r, i);
  if (i < size)
    memcpy(r + i, r, size - i) ;
  ...
}
```

(a) Buggy code snippet.



(b) ExtractFix workflow overview.

Fig. 1. Workflow example from Coreutils

that has a data dependency with the crash location. Step (3) propagates the *CFC* to the fix location along all feasible paths. In this case, the *CFC* is propagated along one path with path constraint $i{<}size$, and *CFC* remains unmodified. Step (4) synthesizes a patch f to replace the *for*-condition. To completely fix the bug, we should ensure `size≤2*i` is always satisfied after applying f. In this case, the synthesizer gives `i <= size/2`. Thus, the program can be patched as follows:

```
- for (i = 3; i < size / 2; i *= 2)
+ for (i = 3; i <= size / 2; i *= 2)
```

The resulting patch is equivalent to the developer patch. In contrast, test-driven program repair approaches may produce over-fitting patches. For example, the following patch fixes the bug for `size=13`, but does not generalize to other crashing inputs such as `size=7`:

```
+ for (i = 3; i < size / 2 || i == 6; i *= 2)
```

3 BACKGROUND ON SYNTHESIS

Given a set of specifications, program synthesis generates a program satisfying the specifications. Program synthesis is formalized to be a second-order constraint solving problem in the recent work on *SE-ESOC* [29]. We build our program synthesizer on top of the approach proposed by SE-ESOC. Given a set of components C, this approach first constructs the set of terms and represents them via a tree. Specifically, each leaf of the tree corresponds to components without input, and intermediate node has as many subnodes as the maximal number of inputs of a component. For each node i with sub-node $\{i_1,i_2,...,i_k\}$, the output and inputs are represented by out_i and $\{out_{i_1},out_{i_2},...,out_{i_k}\}$, respectively. In addition, boolean variables s_i^j is the j-th selector of node i, which means j-th component is used in this node, F_j represents the semantics of j-th component, and N is the number of nodes in the tree. The well-formedness constraint is encoded as $\varphi_{wfp} := \varphi_{node} \wedge \varphi_{choice}$, such that:

$$\varphi_{node} := \bigwedge_{i=1}^{N} \bigwedge_{j=1}^{|C|} \left(s_i^j \Rightarrow \left(out_i = F_j\left(out_{i_1}, out_{i_2}, ..., out_{i_k}\right)\right)\right) \quad (1)$$

$$\varphi_{choice} := \bigwedge_{i=1}^{N} exactlyOne\left(s_i^1, s_i^2, ..., s_i^C\right) \quad (2)$$

For a node, φ_{node} describes the semantic relations of each node between its output and inputs, where the inputs are the outputs of its sub-nodes. φ_{choice} restricts that only exactly one component is selected inside each node. Using the above encoding, the output of root node represents a function f that connects inputs and outputs of components. Finally, given n input-output pairs $\{\alpha_k, \beta_k \mid$

$1 \leq k \leq n\}$, the synthesis goal is to generate function f by traveling the abstract tree and make $\varphi_{correct}$ satisfied.

$$\varphi_{correct} := \bigwedge_{k=1}^{n} \beta_k = f(\alpha_k) \tag{3}$$

4 METHODOLOGY

Our workflow involves constraint extraction, propagation and patch synthesis. In this section, we present each step in more detail.

4.1 Crash-Free Constraint Extraction

Our workflow begins with a vulnerable program and a single crashing input. The first step is to extract both (1) the *crashing location* (e.g., filename/lineno), and (2) the *crash-free constraint* (*CFC*) representing the condition that was violated and the underlying cause of the crash. For (1), the crash location is extracted according to debugging information when the crash is triggered, which needs us to compile the program with debugging option. For (2), the *CFC* extraction is *template*-based, and is instantiated from the crashing expression/statement. Our tool chain currently considers three basic classes of crash:

(1) *Developer*-induced crashes, i.e., `assert`(C) failure;
(2) *Sanitizer*-induced crashes caused by the program violating a sanitizer-enforced *safety property* (e.g., memory safety, type safety, etc.); or
(3) *Hardware*-induced crashes due to the program executing an illegal operation (e.g., null-pointer access, divide-by-zero, etc.).

A summary of the different kinds of crashes and the corresponding *CFC*-templates are shown in Table 1. Here, the *crash expression* is matched against the corresponding crashing expression/statement from the buggy program, and the *CFC*-template is instantiated accordingly. We choose those templates because they cover the common errors and vulnerabilities in C/C++ programs, e.g. null pointer deference, integer/buffer overflow. In this paper, we restrict to fix the bugs supported by these templates. Our tool can also fix other kinds of bugs by extending the templates.

For Example 2.1, the crashing statement `memcpy(r+i, r, size−i)` is matched against the template from Table 1 using the substitution `p=r+i`, `q=r`, and `s=size−i`. This yields the following *CFC* after substitution and simplification:

$$(\texttt{r+i+size-i} \leq \texttt{r} \vee \texttt{r+size-i} \leq \texttt{r+i}) \equiv (\texttt{size} \leq 0 \vee \texttt{size} \leq 2*\texttt{i})$$

We shall discuss the *CFC* generation in more detail.

4.1.1 User-Assertion/Hardware Constraint Extraction.

The *CFC* for user assertions and hardware-induce crashes is relatively straightforward to generate. Assuming the crash is caused by a *user assertion failure* `assert`(C), the *CFC* can be read directly from the assertion statement itself, i.e., *CFC*=C. Crashes can be caused by hardware faults such as `NULL`-pointer dereference and divide-by-zero are detected using an appropriate *signal handler*, e.g., `SIGSEGV` with `si_addr=0` and `SIGFPE` with `si_code=FPE_INTDIV` respectively. The corresponding *CFC* ensures that the crashing symbolic pointer/divisor is not zero.

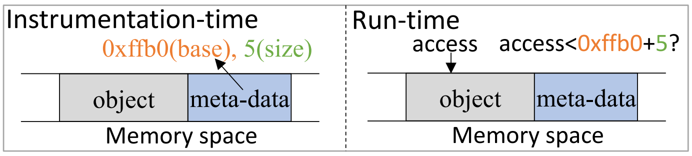
4.1.2 Sanitizer Constraint Extraction.

For our purposes, a *sanitizer* is any dynamic analysis tool that instruments/modifies the program with additional runtime checks enforcing some *safety properties*, such as memory safety, preventing integer overflows, or other undefined behavior avoidance. Typically, sanitizers insert instrumented checks/assertions before relevant operations. For example, as shown in the following figure, the instrumentation of most spatial memory safety sanitizers (a.k.a., bounds-check sanitizers) track *object bounds information* (i.e., the *size* and *base* address of

Table 1. Basic crash classes, crash expressions/statements, and the corresponding *Crash-Free Constraint CFC*-template. We consider seven types of crash: explicit developer assertion violation, sanitizer-induced crash for buffer overflows/underflows, integer overflows, API constraint violation (e.g. non-overlapping regions for `memcpy`), and hardware-induced crashes such as `NULL`-pointer dereference.

Class	Expression	*CFC* Template
developer	`assert`(C)	C
sanitizer	`*p`	`p+sizeof(*p)` $\leq$ *base*(p)+*size*(p) p $\geq$ *base*(p)
	a *op* b	`MIN` $\leq$ a *op* b $\leq$ `MAX` (over $\mathbb{Z}$)
	memcpy(p, q, s)	p+s $\leq$ q $\vee$ q+s $\leq$ p
hardware	*p (for p=0)	p $\neq$ 0
	a / b (for b=0)	b $\neq$ 0

each allocated object) using a disjoint metadata store or related method. At run-time, this metadata is used to look up the object bounds corresponding to the dereferenced pointer, and this pointer is checked against these bounds. If the instrumented check fails, the program is terminated, i.e., "crashes".



Sanitizers can only detect "crashes" on concrete program state, e.g. specific values of *size* and *base* on a certain test. We then symbolize the safety condition that sanitizer enforces by mapping the concrete state back to variables/memory relevant to the crash. For Example 2.1, a sanitizer detects source/destination memory regions overlap when *size=13*. We then generate *CFC* by mapping the concrete value of source/destination back to program variables r and $r+i$, respectively. To map concrete crashing state back to symbolized variables, we extend the meta data by also restoring the corresponding program variable information (e.g. variable name, type) representing *size* and *base*. When the crash is detected, we can simply construct the crash-free constraints using the symbolized program states (program variables). However, in some cases, we may fail to symbolize constraints because some variables used to construct *CFC* are not accessible at the crashing points, i.e. the variables stored in metadata have already been killed at the crashing points. In the general case, we could symbolize the *CFC* using an *extended program state*.

Sanitizer Constraint Language. Some sanitizer-inserted instrumented checks enforce conditions over an *extended state* that is managed by a runtime library or additional instrumentation. This extended state is not part of the original program itself. As such, the sanitizer assertion is over an extended program state that includes the sanitizer runtime. To handle sanitizer-extended state, we allow the generated *CFC* to include functions/types/variables that do not necessarily appear in the original program. For example, in the case of bounds-check sanitizers, we introduce two new abstract functions:

- *base*(p): the base address of the object referenced by p; and
- *size*(p): the size (in bytes) of the object referenced by p.

ALGORITHM 1: Fix localization algorithm

```
Input: A crash location (crashLoc) and an Inter-procedure Control Flow Graph (ICFG)
Output: A set of candidate fix locations (fixLocs)
1  fixLocs := {crashLoc};
2  repeat
3      fixLocsPrev := fixLocs;
4      foreach fixLoc ∈ fixLocsPrev, loc ∈ ICFG − FixLocsPrev do
5          if depends(loc, fixLoc) ∧ dominates(CFG, loc, crashLoc) then
6              fixLocs := fixLocs ∪ {loc};
7          end
8      end
9  until fixLocsPrev = fixLocs;
10 rFixLocs := rank(fixLocs);
11 return rFixLocs;
```

The generated *CFC* will be over these extended functions (see Table 1). Another example is integer-overflow sanitizers, where the generated *CFC* (e.g., `a+b` $\leq$ `MAX`) is over arbitrary precision integers ($\mathbb{Z}$) rather than the original 32bit integer type. For the purpose of *CFC*-generation, we extract the extended-language constraints "as-is", and defer further simplification/handling to the latter stages of our workflow.

4.2 Dependency-based Fix Localization

Once the crash location and *CFC* have been determined, the next step is to decide one (or more) *fix location(s)* where the patch(es) are to be applied. Typically, existing FL approaches, e.g. spectrum-based FL [36], find candidate fix locations by analyzing the execution trace of passing and failing tests. The FL results depend on the quality of the tests, but high-quality tests are not always available. Unlike traditional FL approaches, we make a minimal assumption that only one failing test (exploit) is available, which is a very common scenario when security vulnerabilities are found.

The main intuition of our dependency-based fix localization is that the fix location(s) ought to exhibit a *control* or *data*-dependency with the crash location. Such that, the statement at fix location can influence the truth value of the *CFC*. Just like spectrum-based FL, our second intuition is that the fix location(s) should intersect with the execution path of the crashing test. As a practical realization of the above intuitions, our tool chain uses the *crash location* as the starting point and performs backward *control* and *data*-dependency analysis along with crashing path. Algorithm 1 summarizes the *fix localization* algorithm to decide candidate fix locations. Here, the algorithm takes as input an *Inter-procedure Control Flow Graph* (ICFG) and a *crash location* (*crashLoc*). Since the ICFG may be large in practice, partial ICFG is constructed by considering locations visited by the failing test (exploit) and dependency analysis is performed with the crashing statement as the slicing criterion. The algorithm iteratively builds a set of potential *fix locations* (*fixLocs*) by adding nodes that (1) have a (transitive) dependency with the crash location, and (2) *dominate* the crash location. Finally, the algorithm generates a set of sorted *fix location* candidates, which is ranked according to the distance to the crashing location.

Dependency Closure. Our algorithm also considers the *transitive closure* of static data and control dependencies [40] of the crashing statement to compute potential fix locations. Data dependencies are determined using the standard *def-use-chain* traversal algorithm over a *Single Static Assignment* (SSA) representation of the program. We detect control dependencies using the standard *Control*

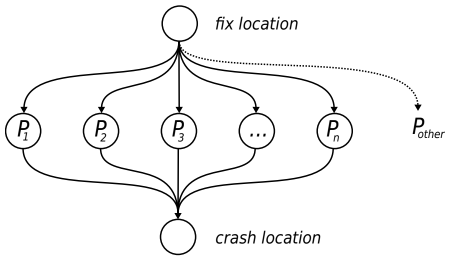


Fig. 2. Illustration of the fix localization algorithm. The algorithm attempts to find a node (*fix location*) that (1) is a dependency of, and (2) dominates the (*crash location*). All paths from the entry point to the crash location must pass through the fix location. There can be more than one path (P_i) between the fix and crash locations. It is allowable that some paths, including loops, from the fix location do not pass through the crash location (P_{other}).

Dependence Graph (CDG) [5] program analysis as part of the LLVM compiler infrastructure. Considering Figure 1a once more, the *for*-condition (line 4) is a control dependency on the assignment statement (`i *= 2`, also line 4), and the crash location (line 7) is data dependent on this assignment. Thus, the *for*-condition is a potential fix location.

Crashing Path and Dominance. The set of all (transitive) data and control dependencies of the crash location can be quite large, leading to many potential fix locations. To reduce the number of potential fix locations, we restrict the fix location(s) should exist somewhere along the concrete path belonging to the original crashing test case. Furthermore, in order to guarantee that the patched program satisfies the *CFC*, our fix localization algorithm only considers statements that *dominate* the crash location—i.e, all paths from the entry point to the crash location must also pass through the fix location, as illustrated in Figure 2. Considering Example 2.1, the *for*-condition (line 4) *dominates* the crash location, since all paths from the entry will visit the *for*-condition at least once. There are usually multiple nodes that dominate the crash location in real-world programs, meaning there are multiple potential fix locations. Note that, there are always at least two nodes that dominate the crash location: the entry point, and the crash location itself.

4.3 Crash-Free Constraint Propagation

A weakest precondition is the least restrictive precondition that will guarantee the postcondition [4]. We consider the problem of backward propagation as finding the weakest precondition *CFC′* at fix location l that necessarily drives program to the crash location and satisfies *CFC*. As shown in [15] (Theorem 9), for all deterministic programs P and any desired post-condition Q: $wp(P, Q)=fwd(P, Q)$, where wp represents the weakest precondition that drives program P to satisfy Q, while fwd is the result generated by forward symbolically executing P from the first statement to the last and substituting the used variables in Q with symbolic variables. In this paper, we follow this approach and use forward symbolic execution to calculate the weakest precondition. Given a fix location l, crash location c, and *CFC*, we perform symbolic execution between l and c, and calculate weakest precondition *CFC′* at l. Our symbolic execution starts concrete execution with a concrete input t until the fix location l. The concrete input t can be the exploit of the vulnerability being fixed, or any test that can drive program to l. From the fix location, we insert symbolic variables and start symbolic execution to explore all the paths Π from fix location l to crash location c.

Symbolic Variables Insertion. At fix location, existing semantics-based repair techniques, e.g. Semfix [35], Angelix [33] and [29], represent the to-be-repaired expression as (either a first-order or a second-order) symbolic variable. Symbolic execution captures the constraint of passing a given test t by exploring alternate paths from the fix location along which the execution of t could be driven in the fixed program. In contrast, in our approach, symbolic execution computes the weakest pre-condition of the crash-free constraint *CFC*, by exploring *all* paths between fix location and crash location. Apart from generating a (second-order) symbolic variable ρ at the fix location capturing the to-be-synthesized expression, we also set the live variables V, on which *CFC* is dependent, as symbolic variables. With these symbolic variables, we can explore and navigate the paths between fix and crash location.

Symbolic execution scope. To avoid exploring irrelevant paths, all the paths that never reach crash location, e.g. P_{other} in Figure 2, are terminated early (whether a path can reach c is determined by analyzing control flow graph). With the help of symbolic variable injection and early termination, the explosion of paths is reduced. Furthermore, we do not suffer from the path explosion issue common in symbolic execution, because fix location is usually close to the crash location.

Constraint collection. After symbolic exploration, we collect the path constraints pc_j for each path $\pi_j \in \Pi$ (all feasible path from l to c). Besides, following each π_j, all the variables used in *CFC* can be represented using the symbolic variables (V and ρ). By replacing the elements in *CFC* with the symbolic representations of V and ρ, we rewrite *CFC* as CFC'_j. Then, $pc_j \Rightarrow CFC'_j$ will be exactly same as the constraint by backward propagating *CFC* from crash location to fix location along path π_j. Consider the following program

$$input\ x,\ i;\ \ if(i{>}0)\ y{=}x{+}1;\ \ else\ y{=}x{-}1;\ \ output\ y;$$

Suppose *CFC* is $(y > 5)$, along the *if-then* branch, we will get the constraint $(i{>}0 \Rightarrow x + 1 > 5)$.

Constraint Simplification (Optional). The propagated constraints may still contain extended sanitizer-supplied functions (e.g., $base(p)/size(p)$) or types (e.g., $\mathbb{Z}$ for integer overflow). There are two basic approaches to handling the extended constraint language: (1) Synthesize the patch "as-is". If necessary, extra functionality can be supplied using a suitable runtime library; or (2) translate the extended constraints into the native language if possible.

Approach (1) is the most general. For example, runtime implementations of the $base(p)/size(p)$ are available using a suitable *library* such as [8], meaning these functions can be used in a patch. The downside is that this introduces an additional dependency on the patched program, which may be undesirable for some applications. The alternative (2) approach is to rewrite the extended constraints back into the native language if possible. For example, using a simple static analysis, our tool searches for a dominating CFG node where the object associated to p is first allocated, e.g., ptr=`malloc`(len). If such a node is found, then our tool can substitute $base(p){=}ptr$ and $size(p){=}len$. This approach is less general than (1) since it depends on a suitable substitution being found.

4.4 Patch Synthesis

After backward propagation of crash-free constraints, patch synthesis is used to rewrite the statement at fix location and guarantee:

$$\{true\}[\ \rho \mapsto f\ ]\{CFC'\}$$

Though our reasoning is performed on partial program (from fix to crash location), the synthesized patch will be also effective for the whole program, because the precondition (*true*) is applied. Once $\{true\}[\rho \mapsto f]\{CFC'\}$ is satisfied, CFC' is guaranteed to be hold under any context.

Instead of satisfying input-output relations as shown in Equation 3, the synthesizer is used to produce a patch satisfying a certain constraint. Suppose Π is the set of feasible paths between fix and crash location, for each path $\pi_j \in \Pi$, the generated patch f should imply CFC'_j under **all** input

ALGORITHM 2: Extension of second-order synthesizer

```
Input: The original buggy expression $e$, the constraint $\varphi_{correct}$
Output: A patch $f$ which satisfies $\varphi_{correct}$
hard := $\varphi_{wfp}$;
soft := $\varphi_{syn}$ ;
patches := $\emptyset$ ;
while |patches| $\leq N$ and (timeout not reached) do
    $f_c$ := pMaxSMT(hard, soft) ;
    $I$ := SMT($\neg\varphi_{correct}[f \mapsto f_c]$) ;
    if $I \neq$ None then
        hard := hard $\wedge\ \varphi_{correct}[V \mapsto I]$ ;
    else
        patches := patches $\cup \{f_c\}$ ;
    end
end
return semSelect(patches) ;
```

space. Then, we change the definition of $\varphi_{correct}$ defined in Section 3 to:

$$\varphi_{\text{correct}} := \bigwedge_{j=1}^{|\Pi|} \left(\left(\rho = f(V) \wedge pc_j \right) \Rightarrow CFC'_j \right) \tag{4}$$

where f represents the to-be-synthesized function and V is the set of variables used by f. For the example 2.1, $\varphi_{correct}$ will be:

$$\varphi_{correct} = (\rho = f(size, i) \wedge \neg\rho \wedge i < size) \Rightarrow size \leq i * 2$$

Since f is a function and the implication should hold **for all** inputs, φ_{correct} is actually a second-order formula. To solve this formula, ExtractFix uses the idea of *second-order solver* [30] to convert φ_{correct} to a first-order formula, and then uses *counter-example guided inductive synthesis (CEGIS) [16]* to find proper patches. By synthesizing f satisfying $\varphi_{correct}$, we can handle all bug-triggering inputs that violate *CFC*′, hence *CFC*.

Though the generated patch makes *CFC* hold, we may still have a wide choice of candidate patches. For fixing the bug in example 2.1, several patches satisfying the $\varphi_{correct}$ (equation 4) could be generated, such as $\{1, i \leq size/2\}$. Obviously, the second one is more likely to be correct. To further improve the quality of patches, the intuition is that correct patch should be similar, both syntactically and semantically, with the original program. To generate "similar" patches, ExtractFix extends *Second-order solver* by further considering the distance between the patched and original program.

The overall workflow of our synthesizer is shown in Algorithm 2, which takes as input the suspicious expression e and $\varphi_{correct}$, and generates a patch f. ExtractFix first generates a patch candidate by solving combined hard and soft constraints using MaxSMT[10] (line 5 of Algorithm 2). The hard constraint is initialized as φ_{wfp} (refer section 3), which ensures the candidate is well-formed. The soft constraint φ_{syn} formulates the syntax distance between buggy expression e and candidate patch. More formally, we build abstract tree T_e for e, and T_c for the patch candidate, and define

$$\varphi_{syn} := \bigcup_{k=1}^{|T_e|} \left\{ T_e^k == T_c^k \right\} \tag{5}$$

where T_e^k (T_c^k) denotes the k-th node of tree $T_e(T_c)$. MaxSMT constructs a patch candidate f_c which strictly satisfies the hard constraint, and satisfies maximum number of soft constraints (shortest distance). The candidate f is then validated by Satisfiability Modulo Theories (SMT) solver [6] to check whether an input that violates $\varphi_{correct}$ exists (line 6). If such a counter-example I exists (line 7-8), the counter-example I is first encoded into first-order logic and then added into hard constraint. Consider the example shown in Figure 1a, in the first iteration, assume $f_c = \lambda i.\ \lambda size.\ i{<}size/2$, then

$$\varphi_{correct} = (\rho = (\lambda i.\ \lambda size.\ i < size/2) \wedge \\ \neg\rho(i, size) \wedge i < size) \Rightarrow size \leq i * 2$$

is violated when $i = 6$ and $size = 13$. Therefore, we add

$$(\rho = f \wedge \neg\rho(6, 13) \wedge 6 < 13) \Rightarrow 13 \leq 12$$

i.e. $f(6, 13) = true$, into the hard constraints. With the refined hard constraints, the candidate f_c generated in the next iteration will ensure $\varphi_{correct}$ must be satisfied under I, i.e. $f_c(6, 13) = true$. Eventually, a plausible patch f_c is thereby generated, which will be added into the *patches* list (line 10). The process continues until *timeout* is reached or we find N plausible patches, where *timeout* and N are defined by users.

Among N plausible patches, the most likely to be the correct one is selected according to its semantic distance to the origin buggy expression e (*semSelect* line 13). Specifically, we (1) generate a set of inputs In that can distinguish plausible patches in terms of their semantics (2) for each $in \in In$, calculate the values of each plausible patch and expression e (3) calculate the value distance between each patch with e (4) select the patch with the shortest distance.

4.5 Multiple-line Fix

The proposed work-flow can be easily extended to support bug-fixing in multiple locations. Fix localization can be generalized as a set of nodes that collectively dominate the crash location, i.e., all paths must go through one of the nodes from the set. Suppose we are introducing patches at location $\{l_1, \ldots, l_n\}$, when propagating *CFC*, multiple second-order variables $\{\rho_1, \ldots, \rho_n\}$ are introduced to represent the to-be-synthesized expressions at $\{l_1, \ldots, l_n\}$, respectively. Correspondingly, the generated *CFC′* will involve multiple second-order variables $\{\rho_1, \ldots, \rho_n\}$. Then, the goal of synthesizer is to generate a set of function $\{f_1, \ldots, f_n\}$ to satisfy:

$$\varphi_{correct} := \bigwedge_{j=1}^{|\Pi|} \left(\Big(\bigwedge_{i=1}^{n} \big(\rho_i = f_i(V_i)\big) \wedge pc_j \Big) \Rightarrow CFC'_j \right) \tag{6}$$

5 EVALUATION

We evaluate the effectiveness and efficiency of ExtractFix and answer the following research questions.

RQ1 Compared with state-of-the-art automated program repair tools, what is the overall effectiveness of ExtractFix in fixing vulnerabilities?
RQ2 Can ExtractFix address the overfitting problem in automated program repair?
RQ3 What is the efficiency of ExtractFix in generating patches?

5.1 Implementation

We have implemented our approach in a tool named ExtractFix, whose architecture is shown in Figure 3. ExtractFix takes as input the vulnerable program and corresponding exploit (test case) to generate patches. ExtractFix is composed of four main components: *constraint extractor*, *fix locator*, *propagation engine* and *patch synthesizer*.

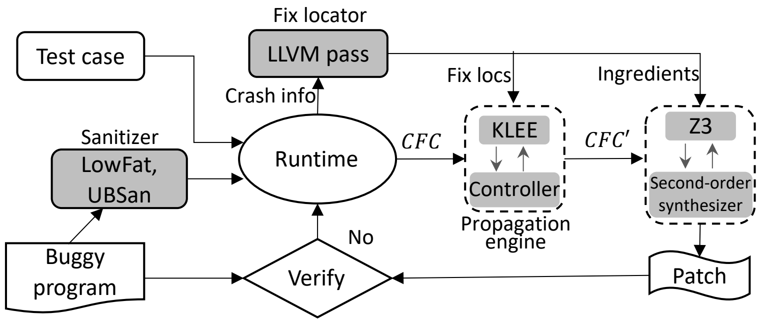


Fig. 3. Architecture of ExtractFix.

Constraint extractor takes as inputs the vulnerable program and exploit, generates crashing location and crash-free constraint *CFC*. The constraint extractor is mainly implemented on top of sanitizers: Lowfat [7, 9] for buffer overflow/underflow and UBSAN [45] for integer overflow. We also considered ASAN [38] for buffer overflow detection, however the LowFat instrumentation more closely corresponds to the *CFC* template, simplifying the extraction process. Although our prototype supports a specific set of defects, other bugs can be supported by integrating new sanitizers and corresponding templates. Once a crash is detected, the concrete crash condition is symbolized into crash-free constraint *CFC* by mapping the concrete value back to program variables. To enable the mapping, the programs should be compiled using clang with debug option.

Fix locator takes as inputs the buggy program and crash information, and produces a set of ranked fix location candidates. The *fix locator* is actually a static analysis tool and is implemented as a LLVM pass [2]. We implement it on top of LLVM because LLVM provides a set of interfaces to generate control flow graph and data dependency graph.

Propagation engine is built on top of KLEE [2]. For the purpose of generating weakest precondition, we extends KLEE in the following two aspects. First, we change the constraint collection by only considering the path constraints between fix and crash location. Second, we early terminate the paths that cannot reach crash location. The execution scope is controlled by *Controller*.

Patch synthesizer is a second-order synthesizer which is implemented according to the approach proposed in [29]. Besides, ExtractFix implements three new features: (1) taking the *CFC* as correctness criterion (2) combining with counter-example guided synthesis and (3) taking into account the distance between patches and original buggy expression. In our implementation of synthesizer, we use Z3 [6] as backend SMT solver.

5.2 Experimental Setup

To evaluate our approach, we choose vulnerabilities from a set of popular applications for ExtractFix to fix by searching the online databases [42–44]. Those databases provide a list of entries, and each of them contains an identification number, a short description of the bug and optional reproducer test case (exploit). We obtain our candidate bugs by searching for the bug types (including buffer-overflow/underflow, integer-overflow, divide-by-zero, null pointer and developer assertion) that our prototype supports. We just consider the bugs reported after 2010 because the earlier bugs is harder to reproduce. Then, we randomly select and manually filter the subjects based on the following four criteria:

[2]LLVM Pass: http://llvm.org/docs/WritingAnLLVMPass.html

Table 2. Subject programs and their statistics

Program	#Vul	Loc	Description
Libtiff	11	81K	library for processing TIFF files
Binutils	2	98K	a set of programming tools for creating and managing binary programs
Libxml2	5	299K	XML C parser and toolkit
Libjpeg	4	58K	C library for manipulating JPEG files
FFmpeg	2	617K	library for processing audio & video
Jasper	2	29K	library for coding & manipulating image
Coreutil	4	78K	GNU core utilities
Total	30	—	—

(1) exploit(s) to trigger the vulnerability are available or exploit(s) can be constructed from the available information;
(2) the target vulnerability has already been fixed by developers so that we have the ground truth on how to fix the bug;
(3) the target application can be compiled into LLVM [20] bitcode and executed by KLEE [2];
(4) the target vulnerability can be reproduced in our environment.

Finally, 30 unique vulnerabilities across seven applications are selected as our benchmark, which includes 16 buffer-overflow/underflow, 4 integer-overflow, 5 divide-by-zero, 3 developer assertion and 2 null pointer dereference. The exploits as well as the instructions to reproduce the bugs are obtained from blogs of researchers, bug reports, exploit databases or the attachments along with patch commit. The selected subjects are across seven applications, and their brief descriptions are given in table 2. Column *Loc* represents their lines of source code, while column *#Vul* shows the number of selected vulnerabilities for each project.

The experiment are directly conducted on these vulnerable applications on a device with Intel Xeon CPU E5-2660 2.00GHz process (56 cores) 64G memory and 16.04 Ubuntu. We set timeout for the symbolic execution and program synthesis as 30 minutes each.

5.3 Experimental Results

5.3.1 *How effective is ExtractFix in fixing vulnerabilities?*

To answer RQ1, we evaluate the effectiveness of ExtractFix in the following three aspects: 1) extracting *CFC* 2) finding correct fix locations and 3) generating patches to fix vulnerabilities. Recall that the vulnerabilities are formalized as violations of constraints, we first evaluate whether ExtractFix can successfully extract such constraints for the given vulnerabilities. For each generated constraint, we verify its correctness by manually investigating the source code and root cause of the vulnerability. Given *CFC*, we then evaluate whether ExtractFix can find the correct fix locations by referring to the developer patches. As our dependency-based fix localization creates a set of ranked candidate fix locations, we retrieve how many candidates we need to inspect until we hit the correct one. We set four levels of correctness: T-1, T-3, T-5 and T-10, where T-N means the correct fix location is hit within top N candidates. Given *CFC* and fix location candidates, we then evaluate the effectiveness of ExtractFix in generating fix, and compare it with existing automated program repair tools: Prophet [28], Angelix [33] and Fix2Fit [12]. Prophet is a search-based automated program repair tool, which ranks patch candidates using a machine learning based approach. Angelix is a state-of-the-art semantic-based program repair tool, which extracts patch requirements from test cases and then directly synthesizes a patch. Fix2Fit proposes to generate

Table 3. Patch generated by ExtractFix

Subject	Vulnerability ID	Type	CFC	FL	Patched	Correct?	Time (m)
	CVE-2016-5321	BO	✓	T-1	✓	Syntactic Equiv.	1.68
Libtiff	CVE-2014-8128	BO	✓	T-1	✓	Semantic Equiv.	2.40
	CVE-2016-5314	BO	✗	-	✗	—	—
	Bugzilla 2633	BO	✓	T-5	✓	Plausible	4.03
	CVE-2016-10094	BO	✓	T-3	✓	Plausible	1.87
	CVE-2016-3186	DL	✓	T-1	✓	Syntactic Equiv.	32
	CVE-2017-7601	IO	✓	T-3	✓	Plausible	2.38
	CVE-2016-9273	BO	✗	-	✗	—	—
	CVE-2016-3623	DZ	✓	T-10	✓	Semantic Equiv.	2.05
	CVE-2017-7595	DZ	✓	T-3	✓	Semantic Equiv.	2.20
	Bugzilla 2611	DZ	✓	T-1	✓	Semantic Equiv.	2.13
Binutils	CVE-2018-10372	BO	✓	T-1	✓	Plausible	16.57
	CVE-2017-15025	DZ	✓	T-3	✓	Semantic Equiv.	36.00
	CVE-2016-1834	IO	✓	T-5	✓	Plausible	5.97
Libxml2	CVE-2016-1839	UO	✗	-	✗	—	—
	CVE-2016-1838	BO	✓	T-1	✓	Plausible	4.12
	CVE-2012-5134	UO	✓	T-3	✓	Syntactic Equiv.	40.83
	CVE-2017-5969	ND	✓	T-1	✓	Syntactic Equiv.	4.30
	CVE-2018-14498	BO	✓	T-3	✓	Plausible	1.22
Libjpeg	CVE-2018-19664	BO	✗	-	✗	—	—
	CVE-2017-15232	ND	✓	T-1	✓	Semantic Equiv.	1.37
	CVE-2012-2806	BO	✓	T-5	✓	Semantic Equiv.	33.26
	CVE-2017-9992	BO	✓	T-3	✓	Semantic Equiv.	9.27
FFmpeg	Bugzilla-1404	IO	✓	T-3	✓	Semantic Equiv.	7.20
	CVE-2016-8691	DZ	✓	T-3	✓	Semantic Equiv.	1.08
Jasper	CVE-2016-9387	IO	✓	T-10	✓	Plausible	1.05
Coreutil	Bugzilla-26545	DL	✓	T-3	✓	Syntactic Equiv.	6.03
	Bugzilla-25003	DL	✓	T-1	✓	Syntactic Equiv.	4.30
	GNUBug-25023	BO	✗	-	✗	—	—
	GNUBug-19784	BO	✗	-	✗	—	—
Total	30	—	24		24	16	(avg) 9.31

BO: buffer overflow; *BU*: buffer underflow; *IO*: integer overflow; *DZ*: divide-by-zero; *DL*: developer assertion; *ND*: null pointer dereference

additional test cases to filter out the overfitted patches. Since Prophet, Angelix and Fix2Fit are all test-driven program repair tools, we run all the three tools with test cases which are composed of 1) exploit that can trigger the vulnerability and 2) available developer tests. Note that, except for one exploit, ExtractFix does not need additional test, developer tests are used to verify the generated patches which is an optional step.

Table 3 shows our evaluation results for each defects. We represent the vulnerability id using its CVE number, bug id in Bugzilla or GNU report, which is shown in column *Vulnerability ID*. Column *Vulnerability type* gives the type of each vulnerability. The effectiveness of ExtractFix is shown in column 4-6, where *CFC* shows whether the constraint is correctly extracted, *FL* represents fix localization results, and *Patched* shows whether the vulnerability is patched by ExtractFix.

Crash-free constraint extraction Out of 30 vulnerabilities, ExtractFix can successfully extract correct constraints for 24 defects, and all of them are correct according to our manual investigation. The results show that our constraint extraction can effectively extract crash-free-constraints, especially for integer overflow, divide-by-zero and developer assertions. We cannot extract correct constraint for six buffer overflow vulnerabilities because the debugging information is ambiguous when symbolizing the condition enforced by sanitizer (the limitation of our prototype).

Fix localization For the cases that we can extract correct constraints, we further evaluate the effectiveness of our fix localization. Out of 24 vulnerability, the correct fix locations of 9 defects are exactly the first candidate T-1 recommended by our fix localization algorithm. The correct fix locations of 19 defects are correctly localized by iterating the top three candidates, while 22 are localized in the top 5 candidates. Instead of purely using the execution trace (e.g. spectrum-based fault localization), our fix localization also considers the program dependency, so that we can effectively localize the faulty statements that may affect the crashing state *CFC*.

Patch generation Once constraints are correctly extracted, ExtractFix then finds potential fix locations and generates patches via constraint propagation and program synthesis. Out of 30 vulnerabilities, ExtractFix can generate 24 patches. Those patches fix the bug by changing condition, modifying the right-value of assignment or inserting an if-guard checker. For instance, to fix the *Libtiff* buffer overflow of CVE-2014-8128, developers add an if-checker at line 571 to break the *while*-loop when *nrows* is equal to 256:

```
571 + if (nrows == 256) break;
```

Instead, ExtractFix fixes the bug by modifying the exit condition of *while*-loop, which is semantically equivalent to developer patch:

```
567 - while (err >= limit)
567 + while (err >= limit && nrows < 256)
```

With this patch, it is guaranteed that the vulnerability cannot be triggered again by the given exploit.

Multi-line fix To fix the *Libjpeg* buffer overflow vulnerability of CVE-2012-2806, ExtractFix generates multiple-line fixes by changing two *for*-loop conditions. All the generated patches can be found in https://extractfix.github.io.

Comparison with state-of-the-art We then compare the repairability of ExtractFix with Prophet, Angelix and Fix2Fit. We cannot run Angelix on some applications because the libraries (e.g. clang 2.9) used by Angelix no longer support the new versions of those applications. We did not run Fix2Fit on Libjpeg since it does not support the compilation using cmake. Prophet fails to build Binutils and FFmpeg. The columns 3-6 of Table 4 represent the number of patches generated by Prophet, Angelix, Fix2Fit and ExtractFix, respectively. Compared with Prophet and Angelix, ExtractFix generates same or more patches for all the applications. Compared with Fix2Fit, ExtractFix generates more patches on Libtiff and Binutils, but less on Coreutils. Fix2Fit uses fuzzing for ruling out patch candidates. In fact, our comparison with Fix2Fit is conservative in favor of Fix2Fit, since Fix2Fit's fuzzing campaigns have an 8 hour timeout, while our program analysis based technique has a timeout of 1 hour (30 minutes for symbolic execution and 30 minutes for program synthesis). Even then ExtractFix generates more plausible patches than Fix2Fit. More importantly, as we will see later, the patches generated by ExtractFix are of significantly higher quality than the patches from Fix2Fit.

Out of 30 vulnerabilities, ExtractFix extracts 24 correct constraints and generate 24 patches. ExtractFix generates more patches than Prophet, Angelix and Fix2Fit.

Table 4. The number of patches and overfitting-free patches generated by Prophet, Angelix, Fix2Fit and ExtractFix

Program	#Vul	Patched?				Equivalence			
		Prophet	Angelix	Fix2Fit	ExtractFix	Prophet	Angelix	Fix2Fit	ExtractFix
Libtiff	11	7	7	7	9	1	0	1	6
Binutils	2	-	-	1	2	-	-	0	1
Libxml2	5	3	0	4	4	0	0	1	2
Libjpeg	4	3	-	-	3	1	-	-	2
FFmpeg	2	-	-	2	2	-	-	1	2
Jasper	2	2	2	2	2	0	0	0	1
Coreutil	4	2	-	3	2	0	-	1	2
Total	30	17	9	19	24	2	0	4	16

5.3.2 *Can ExtractFix address the overfitting problem?*

The generated patch can definitely handle the bug-triggering exploit, but it may overfit to the given exploit. To evaluate patch correctness, we take the developer patch as criteria and examine the patch correctness by manually analyzing developer patch. For each generated patch by ExtractFix, we check its syntactic and semantic equivalence with the developer patch by manually examining if the patch change the program behavior in the same way as developer patch.

In Table 3, column *Correct?* shows the evaluation results. We mark a patch as *Plausible* if it partially fixes the vulnerability or shows different behavior with developer patch. Out of the 24 patches, 16 patches are syntactically or semantically equivalent to developer patches, while 8 of them are plausible patches. A patch is semantically equivalent to developer patches if it fixes the specific crash (enforced by *CFC*) in the same way as developers. Plausible patches exist because (1) the *CFC′* could be incomplete since backward propagation misses some paths between fix and crash location (e.g. paths inside *for*, *while* loop) (2) ExtractFix knows how to completely fix the vulnerability, but has narrow knowledge about the whole program. For instance, an integer overflow CVE-2017-7601 occurs when performing shift operation (1*L*<<*bitssample*) with *bitssample*>=63 (maximal positive signed long integer is $2^{63}-1$). To fix this vulnerability, developer insert an if-checker (*if*(*bitssample*> *16*) *return 0*) before the crash line. With the guidance of crash free constraint *bitssample*<63, ExtractFix fix the bug by inserting *if*(*bitssample*>=*63*) *return 0*. The generated patch completely fixes the integer overflow, but may unintentionally modify the other program behaviors.

We compare ExtractFix with Prophet, Angelix and Fix2Fit for patch quality (syntactically or semantically equivalent to developer patch). The evaluation results are shown in Table 4, where columns 7-10 represent the number of correct patches generated by Prophet, Angelix, Fix2Fit and ExtractFix, respectively. The test suite provided to repair tools is composed of the exploit and all available developer tests (only very few of them can cover the crash line). Prophet and Angelix are test-driven program repair tool, so the quality of patches generated by them highly depends on the quality of test suite. In our setting, the generated patches by these tools can easily overfit the given tests. Specifically, by manually checking the top patches against developer patches, only two patches generated by Prophet is correct and all the patches from Angelix overfit the failing tests. Fix2Fit can filter out some over-fitted patches by test case generation, but the quality of the patches is not high as found by our experiments. Even though Fix2Fit generate 20 patches, only four of them are correct, while others still over-fit the given test suite. ExtractFix generates as many as 16 correct patches. Instead of using tests to guide the patch generation, ExtractFix is guided by constraints which formulate the root cause of the vulnerability.

ExtractFix outperforms Prophet, Angelix and Fix2Fit in generating patches that are both syntactically and semantically equivalent to developer patches.

5.3.3 *How efficient is ExtractFix in generating patches?*

Scalability is one the most challenging problems of symbolic execution, hence semantic-based program repair. In our evaluation, we show that our approach can scale to real-world large applications, e.g. FFmpeg with 617K lines of codes. Meanwhile, the execution time to generate patches is given in Table 3. On average, we only need 9.31 minutes to generate a patch, with maximum of 41 minutes. Our approach is efficient because (1) our symbolic execution is only performed on a small partial program (2) our second-order program synthesis takes into account the distance between patch candidates with original expression and first evaluates candidates that are close to original expression.

ExtractFix can scale to large programs, such as FFmpeg. On average, it takes 9.31 minutes to generate patches.

5.4 Threats to Validity

Internal Validity The main threat to internal validity is that ExtractFix performs backward propagation via symbolic execution which may miss some paths and result in incomplete constraint propagation. Fortunately, we only perform symbolic execution on a very small part of program. Another threat to internal validity is that we derive our *CFC* templates from frequently reported bugs and vulnerabilities, we note that our set of templates is not exhaustive. By extending *CFC* templates, ExtractFix can easily support fixing other kinds of bugs/vulnerabilities. The last internal threat is that we perform manual inspection of the experimental results which might be error-prone. To mitigate this, two authors of the paper double-checked the generated patches.
External Validity The main threat to external validity is that our selection of subjects may not generalize to other programs. We cannot evaluate ExtractFix on dataset used in [14, 23, 41], because FootPatch fixes resource/memory leak (C/C++) and null pointer dereference (Java), large part of defects in ManyBugs are logic bugs, and the datasets (exploits) used by Senx are not open available. Instead, we evaluate ExtractFix on a set of real programs and real CVEs to show its usability. In future, it may be worthwhile to evaluate our approaches on more relevant CVEs and bugs.

6 RELATED WORK

In this section, we discuss the approaches that generate patches via semantic analysis and address the over-fitting problem in program repair. For a general summarization of program repair techniques, the readers could refer to the surveys [13, 34].

Semantic Program Repair Semantics-based techniques like SemFix [35], Nopol [51], DirectFix [32], Angelix [33] and JFIX [21] generate patches in two steps. First, they formulate the requirement to pass all given tests as constraints for the identified program statements. Second, they synthesize a patch for these statements based on the inferred constraints. This type of approach is related to ExtractFix because these approaches also involve constraint extraction and patch synthesis. Semantics-based techniques extract constraints representing partial specifications to pass the given tests. The inferred specification cannot be guaranteed to generalize to inputs outside the test-suite. In contrast, the constraints extracted by ExtractFix represent the condition that was violated and the underlying cause of the crash. Therefore, ExtractFix can alleviate the over-fitting problem in automated program repair by generating patches that generalize beyond the given tests. Specifically, ExtractFix only needs a single exploit trace to generalize the vulnerability, where existing semantic repair techniques usually need a test-suite.

Patch Ranking One way of addressing over-fitting in program repair is to rank patches according to statistical information learned from code repositories. Typical approaches learn from existing patches [22, 28, 37], existing source code [49], or both [17, 46] to rank the patches in the order of likelihood to be correct. On the other hand, Xiong, et al. [48] propose to filter out the patches based on syntactic and semantic distance between patched and original program. Since these approaches are based on statistical information or heuristics, there is no guarantee that the generated patches can be generalized beyond tests. In contrast, our approach extracts crash-free constraints and ensures the constraint is satisfied on all tests.

Patch Filtering Several approaches [12, 47, 52] generate new test inputs to test the generated patches, and discard patches that result in crashes. Different from these approaches that perform a-posteriori filtering, our approach directly considers the crash-free constraint in the patch generation and ensures not to generate a patch violating the crash-free constraint.

Static Program Repair Instead of relying on test cases, several approaches propose program repair driven by static analysis and verification techniques. These approaches generate patches for static analysis violation by reasoning in separation logic [41] or learning repair strategies from the wild [1]. These approaches need off-the-shelf static analyzer as oracles which may introduce false positive or false negative. Specifically, the work of [41] generates patches that are guaranteed to satisfy certain heap properties (this covers few common bug types such as memory leaks, resource leaks or null de-reference). Different from our approach that is based on program synthesis to generate a patch, their approach is still search-based, where semantic search [19] is used to identify code snippets that satisfy the desired properties. Furthermore, the entire framework is based on reasoning in separation logic and is used to fix only heap properties.

Reference Implementation In many development scenarios, there exists a reference implementation, and the developers try to be compatible with the reference implementation while optimizing other aspects such as performance. For example, when implementing a Java compiler, OpenJDK is the reference implementation, and other implementations such as Jikes JVM tries to optimize the performance. Based on this observation, [31] proposes program repair with a reference implementation, where the reference implementation serves as an oracle to avoid overfitting. Compared with this approach, our approach does not need a reference implementation.

Customized Program Repair Some program repair approaches are designed to repair a specific type of bugs, such as fixing memory leaks [11, 26] or concurrency bugs [3, 18]. This type of work is related to ours because these approaches also assume the existence of a bug constraint and try to generate patches satisfying the constraint. In contrast, our work does not focus on a specific type of bug but tries to derive a general approach that works for any bug types where a bug constraint can be derived.

Vulnerability Repair The recent work SENX [14] aims to repair vulnerabilities using a combination of predicate generation, patch placement, and patch synthesis. The main difference with SENX is that SenX does not have any analytical understanding of which fix locations are suitable and what fixes to insert, and usually inserts trivial if-conditions to disable the crash at/near the crash location ([14] Table III). Besides, SENX does *not* perform any constraint propagation. In the absence of constraint propagation, SENX relies on heuristics to guide patch generation, which limits it to specific classes of bugs. In contrast, ExtractFix is not limited to certain vulnerabilities. Most of the patches generated by ExtractFix are more general, and modify expressions/statements different from the crash location.

7 CONCLUSION

Over-fitting of generated patches is a key challenge in automated program repair. Over-fitting results from weak specifications, such as a test-suite, driving program repair. In this work, we have sought to tackle over-fitting by directly extracting constraint specifications from an observed vulnerability. Even though the vulnerability is observed on a specific test input (the so-called exploit), our extracted constraint captures the "general reason" behind the vulnerability via symbolization. By propagating the extracted constraint from the crash location to other potential fix locations, we generate fixes via fix localization and patch synthesis. Our work thus goes beyond test-suite driven repair and provides a workflow and tool for exploring the fix space of common software security vulnerabilities as well. We plan to make our tool available open source for usage by the wider research community.

ACKNOWLEDGMENTS

This research was partially supported by a grant from the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (TSUNAMi project, No. NRF2014NCRNCR001-21) and administered by the National Cybersecurity R&D Directorate.

REFERENCES

[1] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. 2019. Phoenix: automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 613–624.

[2] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[3] Yan Cai and Lingwei Cao. 2016. Fixing deadlocks via lock pre-acquisitions. In *ICSE*. ACM, 1109–1120.

[4] Satish Chandra, Stephen J Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *ACM Sigplan Notices*, Vol. 44. ACM, 363–374.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark F. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. 13 (1991). Issue 4.

[6] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[7] Gregory J. Duck and Roland H. C Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Compiler Construction*. ACM.

[8] Gregory J. Duck and Roland H. C. Yap. 2018. An Extended Low Fat Allocator API and Applications. *CoRR* abs/1804.04812 (2018).

[9] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers.. In *NDSS*.

[10] Zhaohui Fu and Sharad Malik. 2006. On Solving the Partial MAX-SAT Problem. In *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*. 252–265. https://doi.org/10.1007/11814948_25

[11] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *ICSE (1)*. IEEE Computer Society, 459–470.

[12] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *ISSTA*. ACM, 8–18.

[13] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Trans. Software Eng.* 45, 1 (2019), 34–67.

[14] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using Safety Properties to Generate Vulnerability Patches. In *Proceedings of the 40th IEEE Symposium on Security and Privacy.*

[15] Ivan Jager and David Brumley. 2010. Efficient directionless weakest preconditions.

[16] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 215–224. https://doi.org/10.1145/1806799.1806833

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *ISSTA*. ACM, 298–309.

[18] Guoliang Jin, Wei Zhang, and Dongdong Deng. 2012. Automated Concurrency-Bug Fixing. In *OSDI*. USENIX Association, 221–236.

[19] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search (T). In *ASE*. IEEE Computer Society, 295–306.

[20] 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*. IEEE Computer Society, 75.

[21] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 376–379.

[22] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *SANER*. IEEE Computer Society, 213–224.

[23] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256.

[24] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* (2019).

[25] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. Memfix: static analysis-based repair of memory deallocation errors for c. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 95–106.

[26] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. MemFix: static analysis-based repair of memory deallocation errors for C. In *ESEC/SIGSOFT FSE*. ACM, 95–106.

[27] Huarui Lin, Zan Wang, Shuang Liu, Jun Sun, Dongdi Zhang, and Guangning Wei. 2018. PFix: fixing concurrency bugs based on memory access patterns. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 589–600.

[28] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *ACM Symposium on Principles of Programming Languages (POPL)*.

[29] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic Execution with Existential Second-Order Constraints. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM.

[30] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic execution with existential second-order constraints. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 389–399. https://doi.org/10.1145/3236024.3236049

[31] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *ICSE*. 129–139.

[32] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 448–458.

[33] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 691–701.

[34] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17:1–17:24.

[35] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference onSoftware Engineering*. IEEE, 772–781.

[36] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. 1997. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Software Engineering Esec/Fse'97*. Springer, 432–449.

[37] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: effective object oriented program repair. In *ASE*. IEEE Computer Society, 648–659.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}'12)*. 309–318.

[39] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543.

[40] Frank Tip. 1995. A survey of program slicing techniques. *Journal of Programming Languages* 3 (1995). Issue 3.

[41] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE*. ACM, 151–162.

[42] Website. [n.d.]. Bugzilla, http://bugzilla.maptools.org/. Accessed: 2019-07-20.

[43] Website. [n.d.]. CVE, https://bugs.chromium.org/p/oss-fuzz. Accessed: 2019-05-22.

[44] Website. [n.d.]. CVE, https://cve.mitre.org/. Accessed: 2019-05-20.

[45] Website. [n.d.]. UndefinedBehaviorSanitizer, https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed: 2019-07-20.

[46] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *ICSE*. ACM, 1–11.

[47] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*. ACM, 226–236.

[48] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *ICSE*. ACM, 789–799.

[49] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE*. IEEE / ACM, 416–426.

[50] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 512–523.

[51] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[52] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *ESEC/SIGSOFT FSE*. ACM, 831–841.

[53] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 26–36.