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 may produce over-fitting patches, where the patches produced 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 constraint representing the vulnerability with the help of 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 taken from ManyBugs benchmark, real-world CVEs and Google's Open-source-systems OSS Fuzz framework. We believe that our work presents a way forward for the overfitting problem in program repair, by generalizing observable hazards/vulnerabilities (as constraint) from a single failing test or exploit.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; **Software testing and debugging**; • **Security and privacy** → *Software security engineering*.

Additional Key Words and Phrases: Automated Program Repair, Overfitting, Constraint Extraction and Propagation

ACM Reference Format:
Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2020. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2020), 27 pages. https://doi.org/10.1145/3418461

1 INTRODUCTION

Automated program repair [28] 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

*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/2020/1-ART1 $15.00
https://doi.org/10.1145/3418461

a given buggy program P to make it pass a test-suite T—i.e., *test-suite driven program repair*. As the goal is to find change that merely passes the test-suite T, the automatically generated patch may *overfit* the test data, meaning that the patched program P' may still fail on program inputs/tests outside of T [24, 47]. The problem is particularly dangerous when fixing software vulnerabilities. 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 [62]. 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 that over-fit the given test suite 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 [47]. The over-fitted patches may change program behaviors in an unexpected way, e.g. change the original functionality. When fixing a crash or vulnerabilities, the patched program could fix the crash/vulnerabilities on the given tests, but it could still be vulnerable on the inputs outside the given test suite.

1.1 Overall Methodology

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

1.2 Challenges

Our constraint-driven program repair methodology involves several challenges that need to be overcome.

- *Constraint extraction*: The first challenge (*Constraint extraction*) is to extract a *crash-free constraint* or CFC from an observable crash/vulnerability. The observable program failure is a concrete property violation when executing a failing test or exploit. In contrast, CFC should capture the properties that all program inputs must satisfy at the crash location, in order to avoid the vulnerability.
- *Fix localization*: Our second challenge lies in *fix localization* (FL). *Fix localization* finds one (or more) suitable fix location(s). Typically, existing FL approaches, e.g. spectrum-based FL [43], rely on a given high-quality test suite, which is not always available. Often, there is only one test in the form of an exploit. So, it is a challenging task to infer fix locations with only one failing test.
- *Constraint propagation*: After we determine the crash-free constraint (CFC) and fix location, guiding patch generation using the constraint is not straightforward because the fix location could be different from the crash location. For instance, the following code shows a bug where the source and destination of memcpy overlap [1]. The bug is witnessed at line 4, but the correct patch was applied at line 1. We could extract constraint at the crash location (line 4) to ensure source and destination do not overlap, however, the constraint cannot be directly used to guide patch generation at fix location (line 1).

[1] http://www.cplusplus.com/reference/cstring/memcpy

```
1 - for (i = 3; i < size / 2; i *= 2)  // the correct fix location
  + for (i = 3; i <= size / 2; i *= 2)
2     memcpy (r + i, r, i);
3 if (i < size)
4     memcpy (r + i, r, size - i);    // the crash location
```

Since fix location(s) could be different from the crash location, the extracted *constraint* must be *propagated* and transformed to guide patch generation at fix location(s).

- *Patch synthesis*: The final challenge is to use *program synthesis* to generate candidate patches that ensure that the *constraint* is satisfied for all possible inputs.

1.3 Tackling the challenges

To address the above challenges, our workflow begins with the detection of an exploitable vulnerability in the form of a *crash*, i.e., unexpected program termination due to control flow reaching an invalid state. With the help of sanitizers, such as AddressSanitizer (ASAN) [46] or UndefinedBehaviourSanitizer (UBSAN) [53], we could convert vulnerabilities into normal program crash; in the rest of this paper, we generally regard a exploit as a failing test. After witnessing a crash in an exploit, we extract the crash-free constraint or CFC (first challenge) using a template-based approach. According to pre-defined templates, a constraint representation of the violated condition—i.e., the *crash-free constraint*—can then be extracted from either the program itself (e.g. user assertion failure), API documentation, or safety properties enforced by dynamic analysis tools such as *sanitizers*. For example, a buffer overflow can be formalized as a 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 crash.

We address the second challenge (*Fix localization*) by examining program dependencies, instead of purely relying on test suites. Specifically, we take as input one failing test, and use the crash location as a starting point and find candidate fix locations using control/data dependency analysis.

Once the fix locations are determined, to solve the third challenge (*Constraint propagation*), we *propagate* the extracted constraint backward from the crash location to one or more suitable *fix locations* by calculating the *weakest precondition*.

To address the last challenge (*Patch synthesis*), we integrated the second-order program synthesis with counterexample guided inductive synthesis. We synthesize a patch so that the 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 allows the program repair system to decide between single-line and multi-line fixes as shown by experiments. We instantiate the proposed approach in a prototype named ExtractFix.

1.4 Contributions of the paper

The contributions of this paper can be summarized as follows.

- *Conceptual Contribution:* We propose a technique for completely fixing security vulnerabilities which alleviates the well-known overfitting problem in program repair [47]. This is important since today, many security vulnerabilities once detected and reported as CVE remains un-fixed for a significant period. The automated repair can thus reduce the exposure to these un-fixed vulnerabilities.
- *Technical Contribution:* 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 program variables relevant to the crash. Our constraint-based

program repair method has several technical novelties. First of all, we provide an effective constraint/dependency based fix localization instead of relying on the widely used statistical fault localization. Secondly, we are able to synthesize non-trivial patches at locations different from the crash location, owing to having scalable constraint propagation. Last, we extend existing second-order program synthesis methods to generate patches with minimal syntactic/semantic changes.

- *Utilitarian Contribution:* We implement our security vulnerability repair approach in a tool named ExtractFix. We evaluate ExtractFix on a wide range of vulnerabilities from ManyBugs benchmark and real-world CVEs. Evaluation results show that ExtractFix can generate more correct patches than state-of-the-art automated program repair tools. 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 certain illegal states where 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 the operating-system (e.g., illegal memory access), or instrumented check inserted by *sanitizers* to enforce some safety properties. Typical sanitizers, such as AddressSanitizer (ASAN) [46] and UndefinedBehaviorSanitizer (UBSAN) [53], *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 condition that should be satisfied 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 at the crash location, then the same crash cannot be repeated for any program input.

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*). The observable program crash is a concrete property violation when executing a failing test or exploit, while CFC should capture the properties *for all* possible inputs. The CFC is the symbolization or abstraction of the concrete violations. We extract CFC 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 location(s)* will be generated using a *dependency-based fix localization* algorithm. Unlike the widely used spectrum-based fault localization (SBFL) [43], we take one failing test as input, and use the crash location as a starting point and find candidate fix locations using control/data dependency analysis.

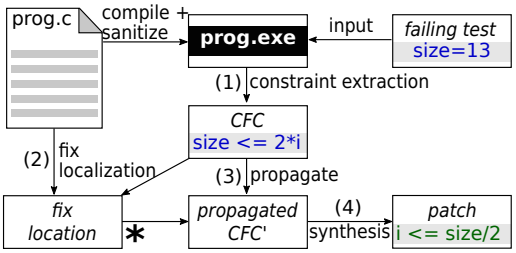
(3) **Constraint Propagation**. The CFC is a constraint over the program state at the crash location. The CFC at the crash location is propagated to a *CFC′* at a given fix location satisfying the following Hoare triple:

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

Here, P represents the program between fix location and crash location. CFC' is the least restrictive (weakest) precondition that will guarantee the postcondition CFC [4]. Finding CFC'

```
1  void fillp (char *r, size_t size){
2    ...
3    r[2] = bits & 255;
4    for (i = 3; i < size / 2; i *= 2)
5      memcpy(r + i, r, i);
6    if (i < size)
7      memcpy(r + i, r, size - i);
8    ...
9  }
```

(a) Buggy code snippet.



(b) ExtractFix workflow overview.

Fig. 1. Workflow example from Coreutils

involves solving CFC-Propagation. For multi-line repair, the approach is generalized and propagation is applied to multiple fix locations.

(4) **Patch Synthesis**. Once the fix location and propagated CFC' have been decided, the next step is to generate patch candidates. Patch synthesis involves rewriting the fix location statement ρ into an alternative f such that the following Hoare triple holds:

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

The generated patch is guaranteed to ensure that *CFC'* is satisfied, meaning that the *CFC* condition at the crash location cannot be violated in the patched program.

Workflow Example. To illustrate our workflow, we consider an example bug 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, the code snippet contains a bug [2]. For certain inputs (e.g., `size`=13), the source and destination regions for the final `memcpy` will overlap— an undefined behaviour under the `memcpy` specification. This bug may cause a 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 a 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

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

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) 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:

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

```
- 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 overfitting patches. For example, the following patch generated by Fix2Fit [11] fixes the bug for `size=13`, but does not generalize to other crashing inputs, e.g. `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* [35]. We build our program synthesizer on top of the approach proposed by SE-ESOC. Given a set of components C, this approach first constructs a set of terms and then represents them as a tree. Specifically, each leaf of the tree corresponds to components without input, and an intermediate node has as many subnodes as the maximal number of inputs of a component. Figure 2 shows a tree with three nodes, and each node is constructed using four components ("x", "y", "+", "-"). The leaf nodes 2 and 3 do not have subnodes, while node 1 has two subnodes since component "+" and "-" takes two inputs. For each node i with sub-node $\{i_1,i_2,...,i_k\}$, its output is represented as out_i, and its inputs is represented by $\{out_{i_1},out_{i_2},...,out_{i_k}\}$ (the output of subnodes). 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. For the tree in Figure 2, with $\{s_1^3, s_2^1, s_3^2\}$ as true, the output of the root node will be $x + y$. The well-formedness constraint is encoded as $\varphi_{wfp} := \varphi_{node} \wedge \varphi_{choice}$, such that:

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

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

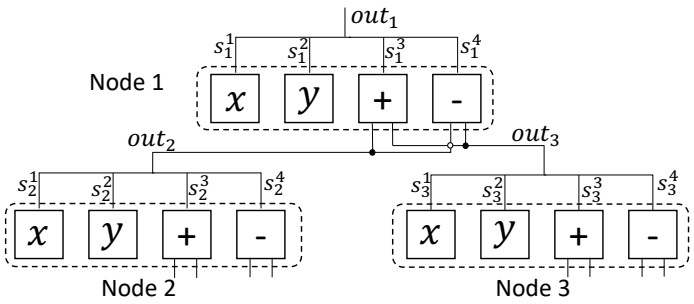


Fig. 2. Encoding with four components ("x", "y", "+", "-") and three nodes.

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 the root node represents a function f that connects inputs and outputs of components. Finally, given n input-output pairs

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 such as buffer overflows/underflows, integer overflows, API constraint violation.

Class	Template ID	Expression	*CFC* Template
developer	T_1	`assert`(C)	C
sanitizer	T_2	`*p`	`p+sizeof(*p)` $\leq$ *base*(p)+*size*(p) `p` $\geq$ *base*(p)
	T_3	`a` *op* `b`	`MIN` $\leq$ `a` *op* `b` $\leq$ `MAX` (over $\mathbb{Z}$)
	T_4	`memcpy(p, q, s)`	`p+s` $\leq$ `q` $\vee$ `q+s` $\leq$ `p`
	T_5	`*p` (for `p=0`)	`p` $\neq 0$
	T_6	`a / b` (for `b=0`)	`b` $\neq 0$

$\{(\alpha_k, \beta_k) \mid 1 \leq k \leq n\}$, the synthesis goal is to generate function f by traversing the abstract tree and satisfying $\varphi_{correct}$, where

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

4 METHODOLOGY

Our workflow for program repair involves constraint extraction, propagation, and patch synthesis. In this section, we discuss each step in more details.

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, meaning that the program must be compiled with debugging enabled (`-g`). For (2), the *CFC* extraction is *template*-based, and is instantiated from the crashing expression/statement. Our repair technique currently considers crashes due to:

(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.);

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 dereference, 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.

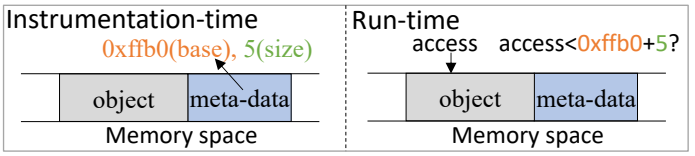
For Example 2.1, the crashing statement `memcpy(r+i, r, size-i)` is matched against the template from Table 1 using the substitution `p=r+i`, `q=r`, and `s=size-i`. This yields the following *CFC* after substitution and simplification:

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

We now discuss the *CFC* generation step in more details.

4.1.1 User-Assertion. The *CFC* for user assertions 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.

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 certain *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 (left part) of most spatial memory safety sanitizers (a.k.a., bounds-check sanitizers) track *object bounds information* (i.e., the *size* and *base* address of each allocated object) using a disjoint metadata store or related method. At run-time (right part of the following figure), this metadata is used to look up the object bounds corresponding to the dereferenced pointer, and this pointer (*access*) is checked against these bounds (*base+size*). If the instrumented check fails, the program is terminated, i.e., "crashes". Crashes can



be caused by hardware failure such as `NULL`-pointer dereference and divide-by-zero are detected using an appropriate santizer or *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.

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 the example in Figure 1a, 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 a concrete crashing state back to symbolized variables, we extend the metadata 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`.

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

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;
```

($\mathbb{Z}$) 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 [43], 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*. We are also looking for fix location(s) which appear on the execution path of the crashing test. As a practical realization of these intuitions, our repair technnique 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 sequence of *fix location* candidates, which are ranked according to the distance to the crash location.

Dependency Closure. Our algorithm also considers the *transitive closure* of static data and control dependencies [48] 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 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 3. 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

The weakest precondition of a formula φ is the least restrictive precondition that will guarantee φ [4]. We consider the problem of backward propagation as finding the weakest precondition CFC' at fix location l that necessarily drives the program to the crash location and satisfies *CFC* at the crash location. As shown in [16] (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 state of variables.

Example 4.1. *Consider the following program* P*:* $(x = x + x; x = x + x; x = x + x)$ *and post-condition* Q*:* $x < 8$*, the weakest precondition to guarantee* Q *is* $wp(P, Q) = \{x < 1\}$*. Similarly, if we set* x *to be a symbolic variable and symbolically execute* P *from the beginning, we would get* $8x < 8$*. That is,* $fwd(P, Q) = \{x < 1\}$.

In this paper, we 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 the 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, or any test that can drive the 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 Variable Insertion. At fix location, existing semantics-based repair techniques, e.g. Semfix [41], Angelix [38] and [35], 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 suite 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. We apply the following transformation schemes to introduce second-order symbolic variable ρ:

- changing the right-hand side of an assignment:

$$x := E; \mapsto x := \rho(v_1, \ldots, v_n);$$

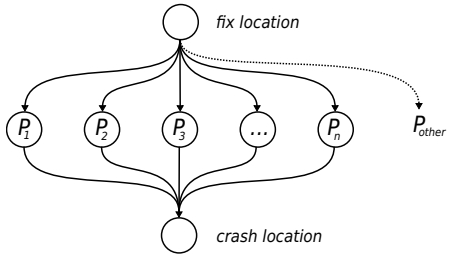


Fig. 3. Illustration of the fix localization algorithm. The algorithm attempts to find a node (*fix location*) that (1) is a dependency of, and (2) dominates the (*crash location*). All paths from the entry point to the crash location must pass through the fix location. There can be more than one path (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}).

- changing a condition:

$$if(E)\{\dots\} \mapsto if(\rho(v_1, \dots, v_n))\{\dots\}$$

- adding an if-guard:

$$S; \mapsto if(\rho(v_1, \dots, v_n))\{S;\}$$

- adding an if-return:

$$insert : if(\rho(v_1, \dots, v_n))\{return\ C;\}$$

where S is statement, E is expression, C is constant and $v_1, \dots, v_n$ are the live variables at the fix location. We use if-return transformation only if the others fail to generate a correct patch, and the error handling code C is generated using Talos [14]. Apart from generating a (second-order) symbolic variable ρ at the fix location (to capture the to-be-synthesized expression) we also set the live variables V (on which *CFC* is dependent) as symbolic variables. We might introduce multiple symbolic variables. If a variable v at the fix location may affect the truth value of *CFC* at the crash location, we will set v as a symbolic variable. This strategy introduces a minimal number of symbolic variables while ensuring that all relevant paths between fix and crash location are explored. 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 3, 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, since fix locations are usually close to the crash location, we can further alleviate the path explosion problem which is common in symbolic execution.

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 the 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 the *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*, 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 to be found.

Note that the weakest precondition calculation inherits the limitation(s) of how symbolic execution is performed. For instance, we may generate incomplete weakest preconditions if there are loops between fixing location and crash location. In our setting, we also inherit the solution from symbolic execution by adding a bound to the number of loop iterations, which may result in incorrect patches as shown in Section 5.

4.4 Patch Synthesis

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

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

Although our reasoning is performed on a 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 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 space. Then, we change the definition of $\varphi_{correct}$ defined in Section 3 to:

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

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* [35] to convert φ_{correct} to a first-order formula, and then uses *counter-example guided inductive synthesis (CEGIS) [17]* 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 the correct patch should be similar, both syntactically and semantically, with the original program. To generate "similar" patches,

ALGORITHM 2: Extension of second-order synthesizer

Input: The original buggy expression e, the constraint $\varphi_{correct}$
Output: A patch f which satisfies $\varphi_{correct}$

```
1  hard := φ_wfp;
2  soft := φ_syn ;
3  patches := ∅ ;
4  while |patches| ≤ N and (timeout not reached) do
5  |   f_c := pMaxSMT(hard, soft) ;
6  |   I := SMT(¬φ_correct[f ↦ f_c]) ;
7  |   if I ≠ None then
8  |   |   hard := hard ∧ φ_correct[V ↦ I] ;
9  |   else
10 |   |   patches := patches ∪ {f_c} ;
11 |   end
12 end
13 return semSelect(patches) ;
```

ExtractFix extends the second-order solver proposed by Mechtaev et al. 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[9] (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 the 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 the hard constraint. Consider the example shown in Figure 1a, in the first iteration, assume $f_c = \lambda i.\ \lambda size.\ i<size/2$, then

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

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.

The patch synthesis of ExtractFix is built on top of SE-ESOC [35]. The difference with SE-ESOC is two-fold. First, we introduce a set of soft constraints (Equation 5) to formulate the distance between original expression and patch candidates. Such that, we can generate patches that are syntactically similar to the original program. Second, SE-ESOC is designed to solve a problem with existential quantifiers, i.e., generate patches to pass **existing** tests. In contrast, the synthesis in ExtractFix generates patches that fix the bug **for all** the valid inputs. Thus, we integrate counterexample guided inductive synthesis into SE-ESOC to make it support solving problems with universal quantifiers, i.e., generate patches to fix the bug under all input space.

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:

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

5 EVALUATION

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

RQ1 What is the overall effectiveness of ExtractFix in fixing vulnerabilities?
RQ2 Compared with state-of-the-art techniques, can ExtractFix alleviate 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 4. ExtractFix takes as input the vulnerable program, exploit (test case) and produces patches. ExtractFix is composed of four main components: *constraint extractor*, *fix locator*, *propagation engine* and *patch synthesizer*.

Constraint extractor takes as inputs the vulnerable program and exploit, generates a crashing location, and a crash-free constraint CFC. The constraint extractor is mainly implemented on top of sanitizers: Lowfat [7, 8] for buffer overflow/underflow and UBSAN [53] for integer overflow, null pointer dereference and etc. 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 the debug option.

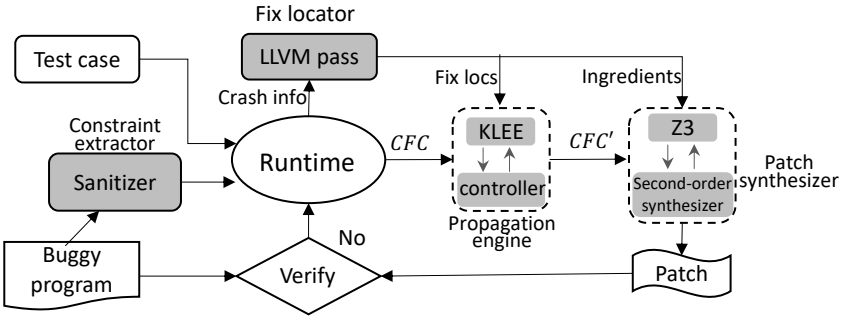


Fig. 4. The architecture of ExtractFix.

Fix locator takes as inputs the buggy program and crash information, and produces a set of ranked fix location candidates. The *fix locator* is a static analysis tool and is implemented as an LLVM pass [3]. We implement it on top of LLVM because LLVM provides a set of interfaces to generate control flow graphs and data dependency graphs.

Propagation engine is built on top of KLEE [2]. For the purpose of generating the weakest precondition, we modify the path exploration of 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 a 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 [35]. 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 the synthesizer, we use Z3 [6] as a backend SMT solver.

5.2 Experimental Setup

Table 2. The 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	—	—

We evaluate our approach on two sets of benchmarks: ManyBugs [26] and our own constructed benchmark. ManyBugs is a C program benchmark suite that is widely used to evaluate automated program repair techniques, such as GenProg [27], Prophet [33] and Angelix [38]. Since we are

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

focusing on vulnerabilities in this paper, we only select bugs that relate to vulnerabilities as our subjects. We therefore select our subjects based on the following criteria:

(1) we only consider bugs related to vulnerabilities, including segmentation fault, buffer overflow/underflow, integer overflow;
(2) the target application can be compiled into LLVM [22] bitcode and executed by KLEE [2];
(3) the target vulnerability can be reproduced in our environment.

We omit two applications of the benchmark (python and fbc) because we could not run these subjects on KLEE. In total, we select 26 defects from three applications: Libtiff, Lighttd, and Php.

Besides ManyBugs, we also constructed an additional vulnerability benchmark suite from a set of popular applications by searching the online databases [50–52]. Those databases provide a list of entries, and each of them contains an identification number, a short description of the bug and optional reproducer (i.e. 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 are harder to reproduce. Then, we randomly select and manually filter the subjects based on the following four criteria:

(1) exploit(s) to trigger the vulnerability is 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 it;
(3) the target application can be compiled into LLVM [22] 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 API 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 application. The main difference between ManyBugs and our own constructed benchmark is that the subjects from ManyBugs include a huge number of test cases, while the subjects in our benchmark only have an exploit and few developer test cases. Note that ExtractFix is designed for working with a few cases.

The experiment is 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. Note that, we do not support parallelism yet. All the results are generated using sequential algorithms.

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 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 the generated constraint, we generate the ground truth of correctness by manually analyzing the source code and root cause of the vulnerability. For instance, we manually analyze the condition that a buffer overflow can be triggered, and check the correctness of CFC. Given CFC, we then evaluate whether ExtractFix can find the correct fix locations by referring to the developer patches. As our dependency-based

Table 3. Evaluation results of ExtractFix. The first part (the first three rows) show the results on ManyBugs benchmark, and the second part present the results on our benchmark.

Application	Defects	CFC	FL (T1/T3)	Patches	Correct Patches	Avg. Time (m)
Libtiff*	5	3	2 / 3	3	2	4.32
Lighttd	3	2	1 / 2	2	1	7.50
Php	18	14	6 / 10	14	9	11.11
Libtiff*	11	9	7 / 8	9	6	5.64
Binutils	2	2	1 / 1	2	1	26.28
Libxml	5	4	3 / 3	4	2	13.80
Libjpeg	4	3	1 / 2	3	2	12.01
FFmpeg	2	2	1 / 1	2	2	8.23
Jasper	2	2	1 / 1	2	1	1.07
Coreutil	4	2	1 / 2	2	2	5.17
Total	56	43	24 / 33	43	28	9.46

* Both Manybugs and our benchmark include the Libtiff program, but the defects in different benchmarks do not overlap.

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. A fix location l is correct if we can generate semantically equivalent patches at l with developer patches. Given CFC and fix location candidates, we then evaluate the effectiveness of ExtractFix in generating fixes, and compare with existing automated program repair tools: Prophet [33], Angelix [38] and Fix2Fit [11]. Prophet is a search-based automated program repair tool, which ranks patch candidates using a machine learning-based approach. In our experiment, we are using the pre-trained model released by the authors of Prophet. Angelix is a state-of-the-art semantic program repair tool, which extracts patch constraints from test cases and then directly synthesizes a patch. Fix2Fit proposes to generate additional test cases to filter out the over-fitted patches. Since Prophet and Fix2Fit are all test-driven program repair tools, we run all those 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 the additional test. As optional post-processing, developer tests could be used to verify the correctness of patches generated by ExtractFix. All the generated CFC, fix locations and patches can be found in https://extractfix.github.io

Table 3 shows our evaluation results. The first part (first three rows) of Table 3 shows the results on ManyBugs benchmark and the second part gives the results on our own constructed subject. The effectiveness of ExtractFix is shown in columns 3-5, where *CFC* shows the number of correctly generated crash-free constraints in each application. *FL* represents fix localization results in a form of (*T1*/*T3*), where *T1* is the number of bugs whose correct fix location is ranked first, and *T3* is the number of bugs whose correct fix location is ranked in the top three candidates. *Patched* shows the number of fixed programs that pass the (single) failure-inducing test. The detailed evaluation result of each defect can be found in Table 4 (ManyBugs) and Table 5 (our benchmark).

Crash-free constraint extraction. Out of 56 vulnerabilities, ExtractFix can successfully extract correct constraints for 43 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 some buffer overflow vulnerabilities and null pointer dereferences because the

Table 4. Patches generated by ExtractFix on ManyBugs Benchmark. Column *Sanitizer* represents the sanitizer used by each defects, where *APISan* is a sanitizer implemented by ourselves to detect violation of API specification, e.g. the destination and source parameters should not overlap in *memcpy*. Column *Template* shows the template id (defined in Table 1) used by each vulnerability, while column *CFC* represents whether we can extract correct crash-free constraints. Column *FL* is the fault localization results, where L-N represents that we need to try *N* fix location candidates until find the correct one. Column *Patched* shows whether we can generate patches to pass the given tests, while *Correct?* gives the correctness of generated patches. *Distance* presents the distance between fix location and crash location.

Subject	ID	Type	Sanitizer	Template	CFC	FL	Patched	Correct?	Distance	Time(m)
Libtiff	207c78a	ND	UBSan	T_5	✗	-	✗	—	—	—
	0a36d7f	BO	Lowfat	T_2	✓	L-1	✓	Sem Equiv.	4	3.44
	ee65c74	IO	UBSan	T_3	✓	L-3	✓	Plausible	10	5.66
	865f7b2	BO	Lowfat	T_2	✗	-	✗	—	—	—
	565eaa2	ND	UBSan	T_5	✓	L-1	✓	Sem Equiv.	2	3.86
Lighttd	1914	BU	Lowfat	T_2	✗	-	✗	—	—	—
	2662	AS	Assert	T_1	✓	L-3	✓	Sem Equiv.	9	8.09
	2786	BO	Lowfat	T_2	✓	L-1	✓	Plausible	7	6.91
Php	5bb0a44e06	ND	UBSan	T_5	✓	L-4	✓	Plausible	10	16.23
	426f31e790	AA	APISan	T_4	✓	L-1	✓	Syn Equiv.	2	14.31
	2a6968e43a	BO	Lowfat	T_2	✓	L-1	✓	Sem Equiv.	2	12.09
	8deb11c0c3	ND	UBSan	T_5	✓	L-1	✓	Plausible	1	10.08
	7f2937223d	ND	UBSan	T_5	✗	-	✗	—	—	—
	2adf58cfcf	ND	UBSan	T_5	✓	L-2	✓	Syn Equiv.	5	9.89
	3acdca4703	ND	UBSan	T_5	✓	L-1	✓	Syn Equiv.	5	9.68
	c2fe893985	ND	UBSan	T_5	✗	-	✗	—	—	—
	93f65cdeac	ND	UBSan	T_5	✗	-	✗	—	—	—
	8d520d6296	ND	UBSan	T_5	✓	L-1	✓	Sem Equiv.	1	7.89
	cacf363957	AA	APISan	T_4	✓	F	✓	Plausible	2	8.96
	c1e510aea8	ND	UBSan	T_5	✓	L-2	✓	Sem Equiv.	4	10.23
	f330c8ab4e	ND	UBSan	T_5	✓	L-5	✓	Sem Equiv.	32	10.24
	1d6c98a136	ND	UBSan	T_5	✓	F	✓	Plausible	2	30.02
	acaf9c5227	ND	UBSan	T_5	✓	L-1	✓	Sem Equiv.	7	5.89
	032bbc3164	BO	Lowfat	T_2	✓	L-2	✓	Sem Equiv.	47	4.30
	1923ecfe25	AA	APISan	T_4	✗	-	✗	—	—	—
	cfa9c90b20	ND	UBSan	T_5	✓	L-1	✓	Plausible	1	5.66
Total	26	—	—	–	19	—	19	12	(avg) 8.1	(avg) 9.6

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

debugging information is ambiguous when symbolizing the condition enforced by sanitizers (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 43 vulnerabilities, the correct fix locations of 24 defects are exactly the first candidate T1 recommended by our fix localization algorithm. The correct fix locations of 33 defects are correctly localized by looking into the top three candidates (T3).

Patch generation. Once constraints are correctly extracted and fix location candidates are determined, ExtractFix then generates patches via constraint propagation and program synthesis. Out of 56 vulnerabilities, ExtractFix can generate 43 patches. Those patches fix the bug by changing conditions, modifying the right-value of assignment or inserting an if-guard checker. For instance,

Table 5. Patches generated by ExtractFix. Column *Sanitizer* represents the sanitizer used by each defects, where *APISan* is a sanitizer implemented by ourselves to detect violation of API specification, e.g. the destination and source parameters should not overlap in *memcpy*. Column *Template* shows the template id (defined in Table 1) used by each vulnerability, while column *CFC* represents whether we can extract correct crash-free constraints. Column *FL* is the fault localization results, where L-N represents that we need to try *N* fix location candidates until find the correct one. Column *Patched* shows whether we can generate patches to pass the given tests, while *Correct?* gives the correctness of generated patches. *Distance* presents the distance between fix location and crash location.

Subject	Vulnerability ID	Type	Sanitizer	Template	CFC	FL	Patched	Correct?	Distance	Time(m)
Libtiff	CVE-2016-5321	BO	Lowfat	T_2	✓	L-1	✓	Syn Equiv.	2	1.68
	CVE-2014-8128	BO	Lowfat	T_2	✓	L-1	✓	Sem Equiv.	5	2.40
	CVE-2016-5314	BO	Lowfat	T_2	✗	-	✗	—	—	—
	Bugzilla 2633	BO	Lowfat	T_2	✓	L-5	✓	Plausible	12	4.03
	CVE-2016-10094	BO	Lowfat	T_2	✓	L-2	✓	Plausible	2	1.87
	CVE-2016-3186	AA	APISan	T_4	✓	L-1	✓	Syn Equiv.	2	32
	CVE-2017-7601	IO	UBSan	T_3	✓	L-1	✓	Plausible	3	2.38
	CVE-2016-9273	BO	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2016-3623	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	2	2.05
	CVE-2017-7595	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	2	2.20
	Bugzilla 2611	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	1	2.13
Binutils	CVE-2018-10372	BO	Lowfat	T_2	✓	F	✓	Plausible	2	16.57
	CVE-2017-15025	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	2	36.00
Libxml2	CVE-2016-1834	IO	UBSan	T_3	✓	F	✓	Plausible	12	5.97
	CVE-2016-1839	BU	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2016-1838	BO	Lowfat	T_2	✓	L-1	✓	Plausible	3	4.12
	CVE-2012-5134	BU	Lowfat	T_2	✓	L-1	✓	Syn Equiv.	2	40.83
	CVE-2017-5969	ND	UBSan	T_5	✓	L-1	✓	Syn Equiv.	2	4.30
Libjpeg	CVE-2018-14498	BO	Lowfat	T_2	✓	L-10	✓	Plausible	3	1.22
	CVE-2018-19664	BO	Lowfat	T_2	✗	-	✗	—	—	—
	CVE-2017-15232	ND	UBSan	T_5	✓	L-1	✓	Sem Equiv.	2	1.37
	CVE-2012-2806	BO	Lowfat	T_2	✓	L-3	✓	Sem Equiv.	10	33.26
FFmpeg	CVE-2017-9992	BO	Lowfat	T_2	✓	L-4	✓	Sem Equiv.	7	9.27
	Bugzilla-1404	IO	UBSan	T_3	✓	L-1	✓	Sem Equiv.	3	7.20
Jasper	CVE-2016-8691	DZ	UBSan	T_6	✓	L-1	✓	Sem Equiv.	5	1.08
	CVE-2016-9387	IO	UBSan	T_3	✓	F	✓	Plausible	5	1.05
Coreutil	Bugzilla-26545	AA	APISan	T_4	✓	L-3	✓	Syn Equiv.	4	6.03
	Bugzilla-25003	AA	APISan	T_4	✓	L-1	✓	Syn Equiv.	2	4.30
	GNUBug-25023	BO	Lowfat	T_2	✗	-	✗	—	—	—
	GNUBug-19784	BO	Lowfat	T_2	✗	-	✗	—	—	—
Total	30	—	—	—	24	—	24	16	(avg)4.0	(avg)9.3

BO: buffer overflow; *BU*: buffer underflow; *IO*: integer overflow; *DZ*: divide-by-zero;
AA: API assert; *ND*: null pointer dereference

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 the developer patch:

Table 6. The number of patches and correct patches generated by Prophet, Angelix, Fix2Fit and ExtractFix. The first part (the first three rows) shows the results on ManyBugs benchmark, and the second part presents the results on our benchmark. In each subject, the tool that produces the most patches and the most correct patches is marked in bold.

Program	#Vul	Total Patches				Correct Patches			
		Prophet	Angelix	Fix2Fit	ExtractFix	Prophet	Angelix	Fix2Fit	ExtractFix
Libtiff	5	2	**3**	**3**	**3**	1	1	1	**2**
Lighttd	3	**2**	**2**	**2**	**2**	0	0	0	**1**
Php	18	10	7	9	**14**	6	4	6	**9**
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	56	31	21	33	**43**	9	5	11	**28**

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

With this patch, it is guaranteed that the vulnerability cannot be triggered again. In our benchmark, once a correct constraint is generated, ExtractFix can always generate a patch.

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.

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 6 represent the number of patches generated by Prophet, Angelix, Fix2Fit and ExtractFix, respectively. Compared with Prophet and Angelix, ExtractFix generates the same or more patches for all the applications. Compared with Fix2Fit, ExtractFix generates more patches on Php, Libtiff, and Binutils, but less on Coreutils. This is because Fix2Fit generates plausible patches by efficiently searching from a large patch space and then uses fuzzing to rule out overfitted patches. 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 56 vulnerabilities, ExtractFix extracts 43 correct constraints and generates 43 patches. ExtractFix generates more patches than Prophet, Angelix and Fix2Fit.

5.3.2 *Can ExtractFix alleviate the overfitting problem?*

The generated patch can 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 the 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 changes the program behavior in the same way as the developer patch.

In Table 3, column *Correct Patch* gives the number or patches that are syntactically or semantically equivalent to developer patches. Out of the 43 patches, 28 patches are syntactically or semantically equivalent to developer patches, while 15 of them are plausible patches. We mark a patch as *Plausible* if it partially fixes the vulnerability or changes program behavior differently compared to the developer patch. 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 avoid triggering the vulnerability, but has narrow knowledge about the intended program behavior from developers. For instance, an integer overflow CVE-2017-7601 occurs when performing shift operation ($1L << bitssample$) with $bitssample >= 63$ (maximal positive signed long integer is $2^{63}-1$). To fix this vulnerability, developers 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. While ExtractFix is designed to alleviate overfitting by completely fixing vulnerabilities, it may still change the program behavior in an unintended way.

We compare ExtractFix with Prophet, Angelix, and Fix2Fit for patch quality. The evaluation results are shown in Table 6, 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. Prophet, Angelix, and Fix2Fit are test-driven program repair tools, so the quality of patches generated by them highly depends on the quality of the test suite. For the defects from ManyBugs, test-driven program repair tools have a higher chance to generate correct patches since there are more available tests. On average, there are around 2.9k available tests for each defect from ManyBugs benchmarks (the first part of Table 6).[4] All the test-driven program approaches generate a number of correct patches. Specifically, on the 26 defects, Prophet, Angelix, and Fix2Fit generate 7, 5, and 7 correct patches, respectively. Even then, ExtractFix generates much more (12) correct patches than all those approaches.

In our constructed benchmark (the second part of Table 6), the available tests are very limited, and only very few tests can cover the crash line. Therefore, 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 are correct and all the patches from Angelix overfit the failing tests. Fix2Fit can filter out some overfitted patches by test case generation, but the quality of the patches is not high as found by our experiments. Out of the 20 patches generated by Fix2Fit, only four patches are correct, while others still overfit the given test suite. In contrast, ExtractFix generates as many as 16 correct patches.

> For bugs that are vulnerabilities supported by ExtractFix, 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 of 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

[4] Around half of the 2.9k tests are irrelevant and will not drive the program to the fix locations. Even then, there are still a considerable number of useful tests in each subject.

given in Table 3. On average, we only need 9.46 minutes to generate a patch, with a maximum of 41 minutes. Our approach is efficient because (1) our symbolic execution is only performed on a small partial program. As shown in Table 3 and 5, the averaged distance between fix and crash location is around 6, with maximum of 47. (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 the original expression.

> ExtractFix can scale to large programs, such as FFmpeg. On average, it takes 9.46 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 the program. What matters is that the incompleteness doesn't seem to have a big impact on the effectiveness of the analysis. 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 whose property violation is sanitizable and expressible as a simple formula. The last internal threat is that we perform a 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 the dataset used in [15, 26, 49], because FootPatch fixes resource/memory leak (C/C++) and null pointer dereference (Java), a large part of defects in ManyBugs are logic bugs, and the exploits and fixes of some datasets (exploits) used by SENX are not available. Instead, we evaluate ExtractFix on a set of real programs and real CVEs to show its usability. In the 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 overfitting problem in program repair. For a general summarization of program repair techniques, the readers could refer to the overview articles [28, 44] or the surveys [13, 40].

Generate and Validate Program Repair Search-based approaches first generate a patch space and then search correct patch via meta-heuristic [27], random search [42], test-equivalence analysis [34] or learning approaches [33]. The correctness of patches is validated using the given test suites. Search-based program repair is able to generate high-quality patches and can easily scale to large programs. The weakness of search-based program repair comes from the incompleteness of test suites. Because of the incompleteness of test suites, the generated patches may overfit the given tests and can break untested functionality. Different from these approaches, ExtractFix uses symbolic analysis and reasoning to generate correct patches beyond tests.

Semantic Program Repair Semantics-based techniques like SemFix [41], Nopol [60], DirectFix [37], Angelix [38] and JFIX [23] 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 specifications lay out the requirement for the patch to pass the given test suite. In contrast, the constraints extracted by ExtractFix represent the underlying cause of the crash and the conditions that should be satisfied to fix vulnerabilities. Therefore, ExtractFix can alleviate the overfitting 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 overfitting in program repair is to rank patches according to statistical information learned from code repositories [39]. Typical approaches learn from existing patches [25, 33, 45], existing source code [58], or both [18, 54] to rank the patches in the order of likelihood to be correct. On the other hand, Xiong, et al. [57] 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 [11, 56, 61] generate new test inputs to test the generated patches, and discard patches that result in crashes. On the other hand, Gao, et al. propose to provide the generated patches to developers for review in an interactive way [12]. 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 [49] or learning repair strategies from the wild [1]. Specifically, the work of [49] 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 [20] is used to identify code snippets that satisfy the desired properties. Furthermore, the entire framework is based on the reasoning in separation logic and is used to fix heap properties only.

Fix Localization in APR Fix localization (FL) techniques determine a set of suspicious buggy statements for fixing, which is one of the key steps in automated program repair. Spectrum-based FL [43, 55] is widely used in existing APR which finds 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. According to a recent study [32], only a subset of bugs can be currently localized by Spectrum-based FL techniques. In contrast, our approach does not rely on complete test suites, and may only require a bug-trigger input. Instead of purely relying on test cases, existing techniques improve the fix localization by further utilizing bug report [21, 31] or deep learning approaches [30]. Our dependency-based fix localization is orthogonal to those approaches. Combining with those approaches may generate better fix localization results. Compared with VFix [59], which localizes buggy statements via value flow analysis for null pointer deference, our approach is more general and can be applied to other kinds of bugs.

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, [36] 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 [10, 29] or concurrency bugs [3, 19]. 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 [15] 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 ([15] 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

Overfitting of generated patches is a key challenge in automated program repair. Overfitting results from weak specifications, such as a test-suite, driving program repair. In this work, we have sought to tackle overfitting 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.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their comments which helped improve the paper. This work was partially supported by the National Satellite of Excellence in Trustworthy Software Systems, funded by National Research Foundation (NRF) Singapore under National Cybersecurity R&D (NCR) programme. This work was partially supported by the National Natural Science Foundation of China (61922003, 61672045) and the Natural Science Foundation of Guangdong Province (Grant No. 2020A1515011494).

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 (ESEC/FSE)*. 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 *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vol. 8. 209–224.

[3] Yan Cai and Lingwei Cao. 2016. Fixing deadlocks via lock pre-acquisitions. In *International Conference on Software Engineering (ICSE)*. ACM, 1109–1120.

[4] Satish Chandra, Stephen J Fink, and Manu Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 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. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13 (1991), 451–490.

[6] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 337–340.

[7] Gregory J. Duck and Roland H. C Yap. 2016. Heap Bounds Protection with Low Fat Pointers. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 132–142.

[8] Gregory J. Duck, Roland H. C. Yap, and Lorenzo Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers.. In *Network and Distributed System Security Symposium (NDSS)*.

[9] Zhaohui Fu and Sharad Malik. 2006. On Solving the Partial MAX-SAT Problem. In *International Conference on Theory and Applications of Satisfiability Testing*. 252–265.

[10] 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 *International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 459–470.

[11] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding program repair. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 8–18.

[12] Xiang Gao and Abhik Roychoudhury. 2020. Interactive Patch Generation and Suggestion. In *Automated Program Repair Workshop*. 2.

[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, Mariana DAngelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 618–635.

[15] 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 (S&P)*. 539–554.

[16] Ivan Jager and David Brumley. 2010. Efficient directionless weakest preconditions. In *Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab*.

[17] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*. 215–224.

[18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 298–309.

[19] Guoliang Jin, Wei Zhang, and Dongdong Deng. 2012. Automated Concurrency-Bug Fixing. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 221–236.

[20] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search (T). In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 295–306.

[21] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *Proceedings of The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[22] 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 (CGO)*. IEEE Computer Society, 75.

[23] 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 (ISSTA)*. ACM, 376–379.

[24] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033.

[25] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 213–224.

[26] 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.

[27] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* (2012), 54.

[28] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (2019).

[29] 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 (ESEC/FSE)*. ACM, 95–106.

[30] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 169–180.

[31] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically generating bug fixes from bug reports. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 282–291.

[32] Kui Liu, Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 102–113.

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

[34] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-equivalence analysis for automatic patch generation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 4 (2018), 1–37.

[35] 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)*. ACM.

[36] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *International Conference on Software Engineering (ICSE)*. 129–139.

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

[38] 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.

[39] Hong Mei and Lu Zhang. 2018. Can big data bring a breakthrough for software automation? *Science China Information Sciences* 61, 5 (2018), 056101.

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

[41] 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 (ICSE)*. IEEE, 772–781.

[42] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. 254–265.

[43] 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 *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Springer, 432–449.

[44] Abhik Roychoudhury and Yingfei Xiong. 2019. Automated program repair: a step towards software automation. *Science China Information Sciences* 62, 10 (2019), 200103.

[45] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: effective object oriented program repair. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 648–659.

[46] 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.

[47] 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 (FSE)*. ACM, 532–543.

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

[49] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *International Conference on Software Engineering (ICSE)*. ACM, 151–162.

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

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

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

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

[54] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *International Conference on Software Engineering (ICSE)*. ACM, 1–11.

[55] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 4 (2013), 1–40.

[56] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 226–236.

[57] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *International Conference on Software Engineering (ICSE)*. ACM, 789–799.

[58] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *International Conference on Software Engineering (ICSE)*. IEEE / ACM, 416–426.

[59] X. Xu, Y. Sui, H. Yan, and J. Xue. 2019. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 512–523.

[60] 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.

[61] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 831–841.

[62] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. 2011. How do fixes become bugs?. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 26–36.