Static Exploration of Taint-Style Vulnerabilities Found by Fuzzing

Bhargava Shastry[1], Federico Maggi[2], Fabian Yamaguchi[3], Konrad Rieck[3] and Jean-Pierre Seifert[1]

[1]TU Berlin, Berlin, Germany
[2]FTR, Trend Micro, Inc., Milan, Italy
[3]TU Braunschweig, Braunschweig, Germany

Abstract

Taint-style vulnerabilities comprise a majority of fuzzer discovered program faults. These vulnerabilities usually manifest as memory access violations caused by tainted program input. Although fuzzers have helped uncover a majority of taint-style vulnerabilities in software to date, they are limited by (i) extent of test coverage; and (ii) the availability of fuzzable test cases. Therefore, fuzzing alone cannot provide a high assurance that all taint-style vulnerabilities have been uncovered.

In this paper, we use static template matching to find recurrences of fuzzer-discovered vulnerabilities. To compensate for the inherent incompleteness of template matching, we implement a simple yet effective match-ranking algorithm that uses test coverage data to focus attention on those matches that comprise untested code. We prototype our approach using the Clang/LLVM compiler toolchain and use it in conjunction with *afl-fuzz*, a modern coverage-guided fuzzer. Using a case study carried out on the Open vSwitch codebase, we show that our prototype uncovers corner cases in modules that lack a fuzzable test harness. Our work demonstrates that static analysis can effectively complement fuzz testing, and is a useful addition to the security assessment tool-set. Furthermore, our techniques hold promise for increasing the effectiveness of program analysis and testing, and serve as a building block for a hybrid vulnerability discovery framework.

1 Introduction

Software exploitation is asymmetric, requiring only a single flaw to compromise a system, and at times a network of systems. The complexity inherent to contemporary software increases their attack surface, and plays into the hands of attackers. Consequently, it is imperative that each and every software module is well analyzed and tested before a release. This is especially true for applications that routinely handle untrusted user input such as network and data parsers. Fuzz testing has been the tool of choice for conducting security assessments of these classes of applications.

Although fuzz testing is effective at uncovering software vulnerabilities, it has two practical limitations. First, fuzzing may encounter coverage bottlenecks such as cryptographic code, and non-atomic comparison operations that limit the test coverage achieved, and impede the discovery of latent vulnerabilities. Second, several code bases do not contain test harnesses for security-critical program APIs, making thorough testing dependent on writing new test cases. Writing test cases is a manual process that requires domain-specific knowledge pertaining to the software under analysis. *Even* well-written unit tests do not necessarily permit a thorough systems evaluation. For example, networking stacks contain asynchronous, and stateful API calls that are invoked in an event-driven fashion. Without a practical set-up that injects the right sequence of messages, it becomes difficult to test these APIs. Having said that, simple pre-existing test cases can provide a starting point for a wider exploration of the codebase.

In this paper, we build on the idea that static analysis can perform a broader search for vulnerable code patterns, starting from a handful of fuzzer-discovered program failures. Our working hypothesis is that *any* readily available fuzzable test harness can be used to bootstrap our analysis, reducing the burden of test writing. Therefore, we begin by fuzzing an existing test harness packaged with a codebase and expect to find a handful of program crashes. Subsequently, our analysis proceeds in three steps. First, we narrow down the root cause of the uncovered crashes using a memory error detector such as AddressSanitizer, falling back to execution slice based fault localization when the fault is not memory-based. Fault localization not only narrows the search for vulnerable code patterns, but also provides syntactic and semantic information about the underlying fault. Second, we automatically generate vulnerability templates

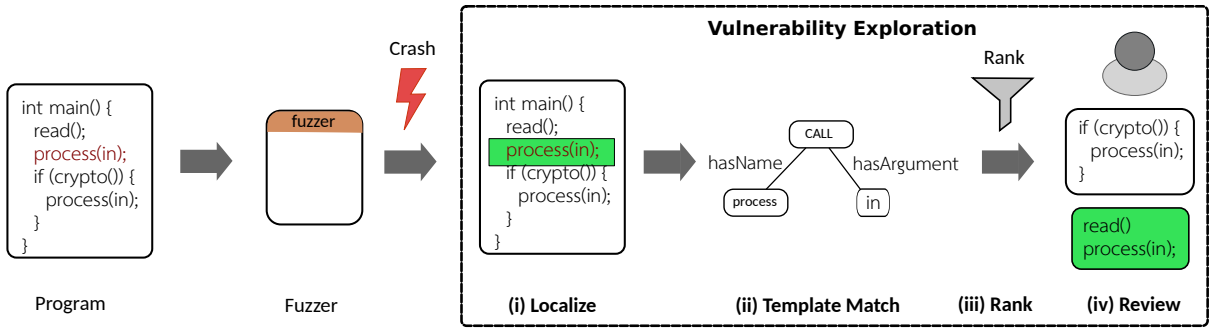

Figure 1: Work-flow of static vulnerability exploration. Templates generated from fault localized code are used to find recurring instances of a fuzzer-discovered vulnerability. The resulting matches are ranked to focus attention on potential recurring vulnerabilities in untested code.

using localized faulty code. Vulnerability templates encode both syntactic, as well as semantic features of the faulty code, making our approach superior to naïve text-based pattern matchers such as `grep`. Third, we rank matching code snippets (returned by template matching) by using fuzzer test coverage data: Matches comprising untested code is ranked higher than those that do not. Such a ranking system helps prioritize manual audit of untested code over code that has already undergone fuzz testing. We use `afl-fuzz`, a contemporary coverage-guided fuzzer, and Clang/LLVM instrumentation and static analysis framework for prototyping our approach.

We evaluate our prototype using a case study of Open vSwitch (OvS), an open-source virtual switch implementation used in data centers. We chose Open vSwitch because (i) it routinely handles untrusted input, and (ii) its existing fuzzable test cases achieve a test coverage of less than 5% providing a low assurance on software security. Results from our case study are promising. Our prototype has uncovered a potential recurring vulnerability in a portion of OvS that lacked a test harness. Moreover, in one instance, template matching has proved to be helpful in flagging a recurring vulnerability that originated in an older release of OvS. This shows that static analysis can not only complement fuzzing, but enable security assessments to be made during software development. To facilitate independent evaluation, we have open-sourced our prototype, that is available at `https://www.github.com/test-pipeline`.

Contributions:

- We present an approach to improve the effectiveness of source code security audit that benefits from both the precise diagnostics of a fuzzer, and the breadth of analysis of a static analyzer.
- We prototype our approach using `afl-fuzz`, a contemporary coverage-guided fuzzer, and the Clang/LLVM compiler toolchain. Our prototype automatically generates vulnerability templates from a fuzzer corpus, ranking the matches returned by template matching based on novelty.
- We evaluate our prototype using a case study of Open vSwitch codebase. Our approached has (i) helped discover one potential vulnerability in a portion of Open vSwitch that lacked a test harness; (ii) facilitated vulnerability checks at an early stage; and (iii) reduced false alarms by 50-100% in most cases demonstrating that coverage-based match ranking is effective in combating false positives.

2 Static Exploration of Vulnerabilities

Contemporary fuzzers and dynamic memory analysis tools have greatly advanced vulnerability detection and re-mediation, owing to their ease-of-use and public availability. Since fuzzers and memory analyzers are invoked at runtime, they require a test harness that accepts user input (usually read from a file or standard input), and invokes program APIs against this input. Therefore, the effectiveness of fuzzing and dynamic memory analysis depends on the availability of test cases that exercise a wide array of program APIs.

In practice, code bases contain test harnesses for only a limited number of program APIs. This means that, even if fuzzing were to achieve 100% test coverage for the set of *existing* test harnesses, it does not lead to 100% *program API* coverage. Furthermore, for networking software, an elaborate test setup is required for thorough testing. Our work seeks to counter practical limitations of fuzz testing using a complementary approach. It builds on the idea that the reciprocal nature of static analysis and fuzzing may be leveraged to increase the effectiveness of source-code security audits. Our key insight is

Listing 1: A representative fuzzer test harness in which two synthetic denial of service vulnerabilities have been introduced by calling the `abort()` function. Fault localized code is shown in red.

```
1  #include <string.h>
2  #include <crypt.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #define CUSTOM() abort()
6  void fuzzable(const char *input) {
7      // Fuzzer finds this bug
8      if (!strcmp(input, "doom"))
9          abort();
10 }
11
12 void cov_bottleneck(const char *input) {
13     char *hash = crypt(input, "salt");
14
15     // Fuzzer is unlikely to find this bug
16     if (!strcmp(hash, "hash_val"))
17         CUSTOM(); // grep misses this
18 }
19
20 // Fuzzer test harness
21 // INPUT: stdin
22 int main() {
23     char buf[256];
24     memset(buf, 0, 256);
25     read(0, buf, 255);
26     fuzzable(buf);
27     cov_bottleneck(buf);
28     return 0;
29 }
```

Listing 2: Template derived from fault localized code (green text) is matched against the test harness source code. Match #2 lists the line of code containing a similar fault pattern that is likely untested by a fuzzer.

```
Template matching:
clang-query> match
 declRefExpr(
           to(
             functionDecl(
                         hasName("abort")
             )
           )
 ).bind("crash")

Match #1:

test.c:9:6: note: "crash" binds here
             abort();
             ^~~~~
test.c:8:5: note: "root" binds here
    if (!strcmp(input, "doom"))
    ^~~~~~~~~~~~~~~~~~~~~~~~~~

Match #2:

 test.c:17:6:  note:  "crash" binds here
             abort();
             ^~~~~
test.c:16:5: note: "root" binds here
    if (!strcmp(hash, "hash_val"))
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
2 matches.
```

that vulnerabilities discovered using a fuzzer can be localized to a small portion of application code from which vulnerability templates may be derived. These templates may then be used to find recurring vulnerabilities that may have been either missed by the fuzzer, or are present in code portions that lack a fuzzable test harness.

Motivating Example We motivate our research using the example code shown in Listing 1. The example contains two synthetic vulnerabilities that are identical: program aborts while parsing (i) the input string literal `doom`; (ii) the input whose cryptographic hash equals the string `hash_val`. The `abort` call following the cryptographic comparison is invoked via an alias called `CUSTOM`. We assume that the fuzzer is able to quickly find the crash due to the string literal comparison (`doom`), but is unlikely to generate the input that satisfies the cryptographic operation. This is a reasonable assumption since hash collisions are highly unlikely. Faced with such a coverage bottleneck, we use the crash discovered by the fuzzer as a starting point for our vulnerability exploration, and proceed in three steps. We first localize the fault underlying the observed crash by computing the set difference of program coverage traces for the crashing and non-crashing runs respectively. Let us assume that the fuzzer corpus contains the crashing input `doom` and it's parent mutation, say `doo`. Using basic block (line) coverage tracing which is fast to obtain, we compute the set difference of the faulty and non-faulty executions to be line 9 (i.e., the `abort` call). Using the localized fault together with the crash stack trace, we then search for similar call sites using an automatically generated AST template. Listing 2 shows the template derived from the line of code containing a call to `abort`, and two matches resulting from template matching. Template matching results in two matches, one of which is the fuzzer-discovered vulnerability, and the other is a recurring instance *hiding* below cryptographic code. Although textual pattern matching for vulnerable code patterns is possible, it breaks down when code properties are crucial to finding a match e.g., call to `abort()` via an alias (line 5 of Listing 1). Finally, we partially rank template matches by checking if the lines of code comprising a match have not been executed by a fuzzer (ranked high), or not (ranked low). In our example, such a ranking would place the undiscovered bug on line 17 of Listing 1 above the bug on line 9 that has been found by the fuzzer.

Leveraging static analysis to complement fuzzing is appealing for two reasons. First, static analysis does not require a test harness, making it well-suited for our problem setting. Second, by taking a program-centric view, static analysis provides a greater overall assurance on software quality or lack thereof. Moreover, since we

leverage concrete test cases to bootstrap our analysis, our vulnerability templates focus on a specific fault pattern that has occurred at least once with a demonstrable test input. This begets greater confidence in the returned matches and a higher tolerance for false positives, from an analyst's point of view.

The proposed vulnerability exploration framework requires a coupling between dynamic and static analysis. We begin by fuzzing a readily available program test case. Subsequently, the following steps are taken to enable static exploration of fuzzer-determined program crashes.

- **Fault localization**: We localize vulnerabilities (faults) reported by the fuzzer to a small portion of the code base using either a dynamic memory error detector such as AddressSanitizer [23], or using differential execution slices. Fault localization serves as an interface for coupling dynamic and static analyses, and facilitates automatic generation of vulnerability templates.

- **Vulnerability Templates**: Using lines of code returned by the fault localization module, together with the crash stack trace, we automatically generate vulnerability templates. The templates are encoded using code properties based on a program abstraction such as the abstract syntax tree (AST). Template matching is used to for finding potentially recurring vulnerabilities.

- **Ranking Matches**: We rank matches returned by template matching before it is made available for human review. Matches comprising lines of code not covered by fuzzing are ranked higher than those that have already been fuzzed.

- **Validation:** Finally, we manually audit the results returned by our analysis framework to ascertain if they can manifest as vulnerabilities in practice.

2.1 Fault Localization

Although a program stack trace indicates where a *crash* happened, it does not necessarily pin-point the root-cause of the failure. This is because, a failure (e.g., memory access violation) manifests much after the trail of the faulty program instructions has been erased from the active program stack. Therefore, fault localization is crucial for templating the root-cause of a vulnerability.

We localize a fuzzer-discovered program failure using a memory detector such as AddressSanitizer [23]. AddressSanitizer is a dynamic analysis tool that keeps track of the state of use of program memory at run time, flagging out-of-bounds reads/writes at the time of occurrence. However, AddressSanitizer cannot localize failures *not* caused by memory access violations. For this reason, we additionally employ a differential execution slicing [1] algorithm to localize general-purpose defects.

Algorithm 1 Pseudocode for execution slice based fault localization.

```
1: function OBTAIN-SLICE(Input, Program)
2:     ▷ Slice generated using coverage tracer
3:     Return lines executed by Program(Input)
4:
5: function OBTAIN-DICE(Slice1, Slice2)
6:     dice = Slice1 - Slice2
7:     return dice
8:
9: function LOCALIZE-FAILURE(Fault − Input, Program, Fuzz − Corpus)
10:    fault-slice = obtain-slice(Fault − Input, Program)
11:    nonfault-input = obtain-parent-mutation(Fault − Input, Fuzz − Corpus)
12:    nonfault-slice = obtain-slice(nonfault − input, Program)
13:    fault-dice = obtain-dice(fault-slice, nonfault-slice)
14:    return fault-dice
```

Agrawal et al. [11] first proposed the use of differential execution slices (that the authors named execution dices) to localize a general-purpose program fault. Algorithm 1 shows an overview of our implementation of this technique. First, the execution slice for a faulty input is obtained ($fault - slice$, line 10 of Algorithm 1). Second, the fuzzer mutation that preceded the faulty input and did not lead to a failure is determined (line 11), and the execution slice for this input obtained (line 12). Finally, the set difference of the faulty and the non-faulty execution slices is obtained (line 13). This set difference is called the fault dice for the observed failure. We obtain execution slices of a program using the SanitizerCoverage tool [3].

In summary, fault localization helps us localize a fuzzer-discovered vulnerability to a small portion of the codebase. Faulty code may then be used to automatically generate vulnerability templates.

2.2 Vulnerability Templates

Faulty code snippets contain syntactic and semantic information pertaining to a program failure. For example, the fact that dereference of the `len` field from a pointer to `struct udp` leads to an out-of-bounds memory access contains (i) the syntactic information that `len` field dereference of a data-type `struct udp` are potentially error-prone; and (ii) the semantic information that tainted input flows into the `struct udp` type record, and that appropriate sanitization is missing in this particular instance. Therefore, we leverage both syntactic, and semantic information to facilitate static exploration of fuzzer-determined program crashes.

[1] An execution slice is the set of source lines of code/branches executed by a given input.

Syntactic and semantic templates are derived from localized code snippets, and the crash stack trace. Syntactic templates are matched against the program's abstract syntax tree (AST) representation, while semantic templates against the program's control flow graph (CFG) representation. In the following, we briefly describe how templates are generated, and subsequently matched.

Syntactic Templates Syntactic templates are matched against the program abstract syntax tree (AST). They may be formulated as functional predicates on properties of AST nodes. We describe the process of formulating and matching AST templates using an out-of-bounds read in UDP parsing code of Open vSwitch v2.6.1 that was found by afl-fuzz and AddressSanitizer.

Listing 3 shows the code snippet responsible for the out-of-bounds read. The faulty read occurs on line 636 of Listing 3 while dereferencing the `udp_header` struct field called `udp_len`. The stack trace provided by AddressSanitizer is shown in Listing 4. In this instance, the fault is localized to the function named `check_l4_udp`. Post fault localization, a vulnerability (AST) template is derived from the AST of the localized code itself.

Listing 3: Code snippet from Open vSwitch v2.6.1 that contains a buffer overread vulnerability in UDP packet parsing code.

```
624 static inline bool
625 check_l4_udp(const struct conn_key *key,
626              const void *data, size_t size,
627              const void *l3)
628 {
629   const struct udp_header *udp = data;
630
631 -  // Bounds check on data size missing
632 -  if (size < UDP_HEADER_LEN) {
633 -    return false;
634 -  }
635
636 -   size_t udp_len = ntohs(udp->udp_len);
637
638   if (OVS_UNLIKELY(udp_len < UDP_HEADER_LEN ||
639                    udp_len > size)) {
640     return false;
641   }
642   ...
643 }
```

Listing 6 shows the AST fragment of the localized faulty code snippet, generated using the Clang compiler. The AST fragment is a sub-tree rooted at the declaration statement on line 636, that assigns a variable named `udp_len` of type `size_t`, to the value obtained by dereferencing a struct field called `udp_len` of type `const unsigned short` from a pointer named `udp` that points to a variable of type to `struct udp_header`. Using the filtered AST fragment, we use AST template matching to find similar declaration statements where `udp_len` is dereferenced. The templates are generated by automatically parsing the AST fragment (as shown in Listing 6), and creating Clang libASTMatcher [1] style functional predicates. Subsequently, template matching is done on the entire codebase. Listing 7 shows the generated template and the matches discovered.

Listing 4: Stack trace for the buffer overread in UDP packet parsing code obtained using AddressSanitizer.

```
1  ================================================
2  ==48662==ERROR: AddressSanitizer:
3  heap-buffer-overflow on address 0x60600000ef3a at
4   pc 0x0000005fd716 bp 0x7ffddc709c70
5   sp 0x7ffddc709c68
6
7  READ of size 2 at 0x60600000ef3a thread T0
8      #0 0x5fd715 in check_l4_udp
9      lib/conntrack.c:636:33
10     #1 0x5fcdcb in extract_l4
11     lib/conntrack.c:903:29
12     #2 0x5f84bd in conn_key_extract
13     lib/conntrack.c:978:13
14     #3 0x5f78c4 in conntrack_execute
15     lib/conntrack.c:304:14
16     #4 0x56df58 in pcap_batch_execute_conntrack
17     tests/test-conntrack.c:186:9
18     ...
19 ================================================
```

AST templates are superior to simple code searching tools such as `grep` for multiple reasons. First, they encode type information necessary to filter through only the relevant data types. Second, they are flexible enough to mine for selective code fragments, such as searching for `udp_len` dereferences in binary operations in addition to declaration statements only.

Listing 5: Match returned using automatically generated AST template shows a potentially recurring vulnerability in Open vSwitch 2.6.1. This new flaw was present in the portion of OvS code that lacked a test harness and was found during syntactic template matching.

```
538 static void
539 pinctrl_handle_put_dhcpv6_opts(struct dp_packet
540  *pkt_in, struct ofputil_packet_in *pin,
541  struct ofpbuf *userdata, struct ofpbuf
542 *continuation OVS_UNUSED)
543 {
544 ...
545     // Incoming packet parsed into udp struct
546     struct udp_header *in_udp =
547                          dp_packet_l4(pkt_in);
548 ...
549     // Dereference missing bounds checking
550     size_t udp_len = ntohs(in_udp->udp_len);
551
552 ...
553 }
```

Listing 5 shows one of the matches discovered (see Match #3 of Listing 7). In the code snippet shown in Listing 5, the OVS controller function named `pinctrl_handle_put_dhcpv6_opts` handles an incoming DHCP packet (containing a UDP packet) that is assigned to a pointer to `struct udp_header`, and subsequently dereferenced in the absence of a bounds-check on the length of the received packet. This is one of the bugs found using syntactic template matching that was reported upstream, and subsequently patched by the ven-

Listing 6: AST of the localized fault that triggers an out-of-bounds read in UDP packet parsing code. AST nodes of interest are shown in green.

```
  |-DeclStmt 0x3232120 <line:636:5, col:41>
  | `-VarDecl lib/conntrack.c:636:12 used udp_len 'size_t':'unsigned long' cinit
...
...
  |             |         `-MemberExpr 0x32318f0 <lib/conntrack.c:636:28, col:33>
'const ovs_be16':'const unsigned short' lvalue -> udp_len 0x3102ae0
  |             |           `-ImplicitCastExpr 0x32318d8 <col:28> 'const struct udp_header *'
<LValueToRValue>
  |             |             `-DeclRefExpr 0x32318b0 <col:28> 'const struct udp_header *'
lvalue Var 0x3231680 'udp' 'const struct udp_header *'
```

Listing 7: AST template matching and its output. The code snippet surrounding match #3 is shown in Listing 5.

```
============================== Query ==============================
let member memberExpr(allOf(hasDeclaration(namedDecl(hasName("udp_len"))),
hasDescendant(declRefExpr(hasType(pointsTo(recordDecl(hasName("udp_header"))))))))

m declStmt(hasDescendant(member))

============================== Matches ==============================
Match #1:
ovn/controller/pinctrl.c:635:5: note: "root" binds here
    out_ip6->ip6_ctlun.ip6_un1.ip6_un1_plen = out_udp->udp_len;
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Match #2:
tests/test-conntrack.c:52:9: note: "root" binds here
        udp->udp_src = htons(ntohs(udp->udp_src) + tid);
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Match #3:
ovn/controller/pinctrl.c:550:5: note: "root" binds here
    size_t udp_len = ntohs(in_udp->udp_len);
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
3 Matches.
```

dor [10]. Moreover, this match alerted the OvS developers to a similar flaw in the DNS header parsing code.

To be precise, vulnerability templates need to encode both data and control flow relevant failure inducing code. Otherwise, explicit sanitization of tainted input will be missed, leading to false positives. To this end, we augment syntactic template matching with semantic (control and data-flow) template matching.

Semantic Templates Control and data-flow templates encode semantic code properties needed to examine the flow of tainted input. However, since each defect is characterized by unique control and data-flow, semantic templates are harder to automate. We remedy this problem by providing *fixed* semantic templates that are generic enough to be applied to any defect type.

We parse the program crash stack trace to perform semantic template matching. First, we determine the function in which the program fails (top-most frame in the crash trace), and generate a template to match other callsites of this function. We call this a *callsite* template. Callsite templates intuitively capture the insight that, if a program failure manifests in a given function, other calls to that function demand inspection. Second, for memory access violation related vulnerabilities, we determine the data-type of the variable that led to an access violation, and assume that this data-type is *tainted*. Subsequently, we perform taint analysis on this data-type terminating at pre-determined security-sensitive sinks such as `memcpy`, `strcpy` etc. We call this a *taint* template. Taint templates provide insight on risky usages of a data-type that is known to have caused a memory access violation. Callsite and taint templates are matched against the program control flow graph (CFG). They have been implemented as extensions to the Clang Static Analyzer framework [2].

2.3 Match Ranking

Matches returned using static template matching may be used to (in)validate potentially recurring vulnerabilities in a codebase. However, since vulnerability templates over-approximate failure-inducing code patterns, false positives are inevitable. We remedy the false-positive problem using a simple yet practical match ranking algorithm.

Algorithm 2 presents the pseudocode for our match ranking algorithm. The procedure called `RANK-MATCHES`

accepts the set of template matches (denoted as *Matches*), and the set of program functions covered by fuzz testing (denoted as *Coverset*) as input, and returns a partially orders list suitable for manual review. For each match, we apply a ranking predicate on the program function in which the match was found. We call this function, the *matching unit*. The ranking predicate (denoted as the procedure *isHigh*) takes two input parameters: the matching function name, and the *Coverset*. Under the hood, *isHigh* simply performs a test of set membership; it checks if the matching unit is a member of the coverset, returning `True` if it is a member, `False` otherwise. All matching units that satisfy the ranking predicate are ranked high, while the rest are ranked low. The ranked list is returned as output.

Our ranking algorithm is implemented in Python using a hash table based data structure. When the coverset is given, ranking a match takes $O(1)$ on average, and $O(n)$ in the worst case, where n is the number of functions in the coverset. On average, the time to rank all matches grows linearly with the number of matches. This is really fast in practice e.g., in the order of a few milliseconds (see Table 3).

Although afl encodes program coverage information internally, the encoded information is not at the source code level. For this reason, our prototype leverages *GCov* [9], a publicly available source code level program coverage tracing tool, for obtaining the coverset of test inputs in the fuzzer corpus. Although our prototype currently uses function as a matching unit, it may be suitably altered to work at the level of source line of code (basic blocks). However, there is a trade-off between coverset (and matching unit) granularity (function vs. basic block) and the run time for obtaining the coverset. Function level tracing is fast but can lead to untested bugs being incorrectly ranked low (e.g., buggy untested line of code in a tested function); basic block level tracing is relatively slower but it eliminates false negatives. For our prototype implementation, we have favored lower run time over ranking soundness in view of the thousands of test cases that fuzzer corpora usually contain. Indeed, our evaluation shows that this is a reasonable trade-off. However, we plan to implement a granularity switch that will permit the user to switch to basic block tracing at a modest run time cost.

Validation Although match ranking helps reduce the burden of false positives, it does not eliminate them entirely. Therefore, we rely on manual audit to ascertain the validity of analysis reports. Nonetheless, our approach focuses attention on recurrences of demonstrably vulnerable code patterns, thereby reducing the extent of manual code audit.

Algorithm 2 Pseudocode for ranking statically explored vulnerability matches.

```
1: function ISHIGH(Matching − unit, Coverset)
2:
3:     for each m − unit in Coverset do
4:         if m − unit == Matching − unit then return True
5:     return False
6:
7: function RANK-MATCHES(Matches, Coverset)
8:     RHigh = ∅
9:     RLow = ∅
10:
11:    for each match in Matches do
12:        if isHigh(match, Coverset) then
13:            RHigh += match
14:        else
15:            RLow += match
16:    return (RHigh, RLow)
```

3 Case Study: Open vSwitch

We evaluated our approach on multiple versions of Open vSwitch, an open-source virtual switch used in data centers. We chose Open vSwitch for evaluation because (i) it is a good representative of production code; (ii) it has insufficient test harnesses suitable for fuzzing, resulting in program edge coverage of less than 5%.

Our evaluations were performed using afl-fuzz for fuzzing, AddressSanitizer for fault localization, falling back to our implementation of differential slice-based fault localization, and our implementation of static template generation, matching, and ranking algorithms. Experiments were carried out on a 64-bit machine with 80 CPU threads (Intel Xeon E7-4870) clocked at 2.4 GHz, and 512 GB RAM.

Fuzzing and Fault Localization Using the baseline fuzzer, we discovered multiple out-of-bounds reads and assertion failures in packet parsing code in Open vSwitch. All the discovered flaws were triaged to ascertain their security impact, and subsequently reported upstream and fixed. For each unique vulnerability, we used our fault localization module comprising AddressSanitizer, and differential execution slicing, to determine the lines of code triggering the vulnerability.

Template Matching Using localized code, we automatically generated a template suitable for matching similar code patterns elsewhere in the codebase. For example, the AST snippet shown in Listing 6 was parsed to derive a template for CVE-2017-9264. Subsequently, we used the tool `clang-query` to perform template matching using the derived template. Listing 7 shows the outcome of template matching for one of the bugs comprising CVE-2017-9264. For each vulnerability that the fuzzer discovered, we counted the number of matches (excluding the known vulnerability itself) returned using template matching.

Fuzzer-Discovered Vulnerability	CVE ID	Explored Matches	True Positives
Out-of-bounds read (IP)	CVE-2016-10377 [4]	5	0
Out-of-bounds read (TCP)	CVE-2017-9264 [7]	10	0
Out-of-bounds read (UDP)	CVE-2017-9264	2	1
Out-of-bounds read (IPv6)	CVE-2017-9264	3	0
Remote DoS due to assertion failure	CVE-2017-9214 [5]	22	0
Remote DoS due to unhandled packet	CVE-2017-9263 [6]	34	0
Out-of-bounds read	CVE-2017-9265 [8]	1	0
Total		96	1

Table 1: Summary of static vulnerability exploration carried out on vulnerabilities found by fuzzing Open vSwitch. For each fuzzer-discovered vulnerability, our prototype generate a vulnerability template, and matched it against the entire codebase.

CVE ID	Explored matches	Ranked high (untested)	Reduction in FP (in %)
CVE-2016-10377	5	0	100
CVE-2017-9264	10	0	100
CVE-2017-9264	2	2	0
CVE-2017-9264	3	0	100
CVE-2017-9214	41	17	59
CVE-2017-9263	34	17	50
CVE-2017-9265	1	0	100
Total	96	36	62

Table 2: Effectiveness of our matching ranking algorithm in highlighting untested code, and assisting in fast review of matches.

We used semantic template matching only when syntactic template matching was too broad to capture the code pattern underlying the vulnerability. For example, if a program crash was caused by a failed assertion, syntactic templates (that matched calls to all assertion statements), were augmented with semantic templates (that matched a smaller subset of assertion statements involving tainted data types).

Ranking The returned matches were ranked using our proposed ranking algorithm (see Algorithm 2), and the ranked output was used as a starting point for manual security audit. Matches ranked high were reviewed first. This enabled us to devote more time to audit untested code, than the code that had already undergone testing.

3.1 Analysis Effectiveness

We evaluated the effectiveness of our approach in two ways: Quantifying (i) the raw false positive rate of our analysis; (ii) the benefit of the proposed ranking algorithm in reducing the effective false positive rate after match ranking was done.

To quantify the number of raw false positives, we counted the total number of statically explored matches, and the number of true positives among them. A match was deemed a true positive if manual audit revealed that the tainted instruction underwent no prior sanitization and was thus potentially vulnerable. Table 1 summarizes our findings. Our prototype returned a total of 96 matches for the 7 vulnerabilities found by fuzzing (listed in column 1 of Table 1). Out of 96 matches, only one match corresponding to CVE-2017-9264 was deemed a new potential vulnerability. This was reported upstream and subsequently patched [10]. Moreover, the reported (potential) vulnerability helped OvS developers uncover another similar flaw in the DHCPv6 parsing code that followed the patched UDP flaw.

Our ranking algorithm ranked untested code over tested code, thereby helping reduce the manual effort involved in validating potential false positives. Although it is hard to correctly quantify the benefit of our ranking algorithm in bringing down the false positive rate, we employ a notion of *effective* false positive rate. We define the effective false positive rate to be the false positive rate only among highly ranked matches. This is intuitive, since auditing untested code is usually more interesting to a security analyst than auditing code that has already undergone testing. Table 2 summarizes the number of effective false positives due to our analysis. In total, there

CVE ID	Localization	Syntactic	Semantic	Ranking	Total Run Time	Normalized
CVE-2016-10377	82ms	1.66s	–	63ms	1.80s	0.20x
CVE-2017-9264 (TCP)	84ms	3.20s	–	64ms	3.34s	0.25x
CVE-2017-9264 (UDP)	86ms	4.77s	–	59ms	4.91s	0.37x
CVE-2017-9264 (IPv6)	91ms	4.71s	–	60ms	4.86s	0.36x
CVE-2017-9214	9ms	8.44s	44.17s	60ms	52.67s	5.51x
CVE-2017-9263	9ms	11.88s	44.26s	59ms	57.09s	5.97x
CVE-2017-9265	111ms	5.74s	–	56ms	5.9s	0.62x

Table 3: Run times of fault localization, template matching, and match ranking for all statically explored vulnerabilities in Open vSwitch. The absolute and relative (to code compilation) run times for our end-to-end analysis is presented in the final two columns. A normalized run time of 2x denotes that our end-to-end analysis takes twice as long as code compilation.

were 36 matches (out of 96) that were ranked high, bringing down the raw false positive rate by 62%. Naturally, we confirmed that the single true positive was among the highly ranked matches.

Match ranking helps reduce, but not eliminate the number of false positives. Indeed, 1 correct match out of 36 matches is very low. Having said that, our approach has borne good results in practice, and has helped advance the tooling required for secure coding. The additional patch that our approach contributed to is not the only way in which our approach met this objective. We discovered that the template derived from the vulnerability CVE-2016-10377 present in an earlier version of Open vSwitch (v2.5.0), could have helped eliminate a similar vulnerability (CVE-2017-9264) that was introduced in a later version (v2.6.1), perhaps during software development itself. We manually checked that, had the newly introduced vulnerability been present in the earlier version of Open vSwitch, it would have been flagged by our tool and ranked high. This shows that our approach is suitable for regression testing. Indeed, OvS developers noted in personal communications with the authors that the matches returned by our tooling not only encouraged reasoning about corner cases in software development, but helped catch bugs (latent vulnerabilities) at an early stage.

3.2 Analysis Runtime

We quantified the run time of our tooling by measuring the total and constituent run times of our workflow steps, starting from fault localization, and template matching, to match ranking. Table 3 presents our analysis run times for each of the fuzzer-discovered vulnerabilities in Open vSwitch. Since fault localization was done using dynamic tooling (AddressSanitizer/coverage tracing), it was orders of magnitude faster (ranging between 9–111 milliseconds) than the time required for static template matching. For each fuzzer-discovered vulnerability, we measured the template matching run time as the time required to construct and match the vulnerability template against the entire codebase. Template matching run time comprised between 92–99% of the end-to-end runtime of our tooling, and ranged from 1.8 seconds to 57.09 seconds. Syntactic template matching was up to 4x faster than semantic template matching. This conformed to our expectations, as semantic matching is slower due to the need to encode (and check) program data and control flow in addition to its syntactic properties. Nonetheless, our end-to-end vulnerability analysis had a normalized run time (relative to code compilation time) of between 0.2x to 5.97x. The potential vulnerability that our analysis pointed out in untested UDP parsing code, was returned in roughly a third of the time taken for code compilation of the codebase. This shows that our syntactic analysis is fast enough to be applied on each build of a codebase, while our semantic analysis is more suitable to be invoked during daily builds. Moreover, given the low run time of our analysis, templates derived from a vulnerability discovered in a given release may be continuously applied to future versions of the same codebase as part of regression testing.

4 Related Work

Our work brings together ideas from recurring vulnerability detection, and program analysis and testing. In the following paragraphs, we compare our work to advances in these areas.

Patch-based Discovery of Recurring Vulnerabilities Redebug [16] and Securesync [22] find recurring vulnerabilities by using syntax matching of templates derived from vulnerability patches. Thus, *patched* vulnerabilities form the basis of their template-based matching algorithms. In contrast, we template a vulnerability based

on automatically localized failures, and debug information obtained from fuzzer reported crashes. What makes our setting more challenging is the lack of a reliable code pattern (usually obtained from a patch) to build a template from. As we have shown, it is possible to construct vulnerability templates even in this constrained environment and find additional vulnerabilities even *in the absence* of patches.

Code Clone Detection We are not the first to present a pattern-based approach to vulnerability detection. Yamaguchi et al. [25] project vulnerable code patterns derived from patched vulnerabilities on to a vector space. This permits them to extrapolate known vulnerabilities in current code, thereby permitting the discovery of recurring vulnerabilities.

Other researchers have focused on finding code clones regardless of them manifesting as vulnerabilities [13, 14, 19, 21]. Code clone detection tools such as CP-Miner [20], CCFinder [18], Deckard [17] solve the problem of finding code clones but rely on sample code input to be provided. These tools solve the more general problem of finding identical copies of user-provided code. Although these tools serve as a building block for recurring vulnerability discovery, they require that the user specifies the code segment to be matched. Manual specification of code templates might not be feasible while auditing third-party code. Therefore, we leverage the fuzzer for discovering vulnerable code patterns.

Hybrid Vulnerability Discovery SAGE [15] is a white-box fuzz testing tool that combines fuzz testing with dynamic test-case generation. Constraints accumulated during fuzz testing are solved using an SMT solver to generate test cases that the fuzzer alone could not generate. This is expensive because it requires a sophisticated solver. In a similar vein, Driller [24] augments fuzzing through selectively resorting to symbolic execution when fuzzer encounters coverage bottlenecks. The use of symbolic execution to augment fuzzing is complementary to our approach. In practice, security audits would benefit from both our approach as well as that proposed by prior researchers.

Saner [12] combines static and dynamic analyses towards identifying XSS and SQL injection vulnerabilities in web applications. The authors of Saner use static analysis to capture a set of taint source-sink pairs from web application code, and subsequently use dynamic analysis on the captured pairs to tease out vulnerabilities. Their evaluation on popular PHP applications show that dynamic analysis is able to bring down the number of false positives produced by static analysis, and find multiple vulnerabilities. Like our work, Saner demonstrates that static and dynamic analyses can effectively complement each other. In contrast to Saner, we differ in the order of analyses performed (we perform static analysis driven vulnerability exploration after confirmed taint source-sink pairs have been found), and in the target programming language.

Yamaguchi et al. [26] automatically infer search patterns for taint-style vulnerabilities from source code by combining static analysis and unsupervised machine learning. They show that their approach helps reduce the amount of code audit necessary to spot recurring vulnerabilities by up to 94.9%, enabling them to find 8 zero-day vulnerabilities in production software. Their work is close in spirit to ours. However, we avoid the computational overhead involved in their workflow (building a code property graph, pattern clustering etc.), while retaining their template matching run time. In our framework, fault localization and result ranking run times are almost negligible.

5 Conclusions and Future Work

Fuzzing is a time-tested technique for discovering taint-style vulnerabilities in software. However, fuzzing is mainly limited by test coverage, and the availability of fuzzable test cases. In this paper, we leverage static analysis to perform an exhaustive search by using fuzzer-discovered vulnerabilities as a starting point.

We use fault localization techniques to narrow down the search for vulnerable code patterns. Subsequently, localized code is used to automatically generate vulnerability templates. False positives have been the primary drawback of static analysis tools. As a remedy, we propose a ranking algorithm that brings attention to potential vulnerabilities in untested code.

We evaluate our approach on multiple versions of the Open vSwitch codebase, a popular virtual switch used in data centers. Using static exploration of fuzzer-discovered vulnerabilities, we were able to discover an additional potential vulnerability in untested code. Furthermore, we show that a vulnerability template derived from a dated vulnerability would have helped discover a recurring vulnerability in a later software release. This shows that static vulnerability exploration has the potential to weed out flaws at an early stage of software development. Indeed, our case study highlights the need to complement existing software testing approaches like fuzzing with static analysis.

Our work leaves open multiple avenues for future work. At present, we rely on manual validation of statically discovered faults. This may be complemented using selective symbolic execution tools such as *angr* so that additional diagnostics such as path reachability and concrete test input may be obtained. Orthogonally, the

precision of our templates can be improved by modeling data sanitization functions more precisely.

Acknowledgements. We would like to thank Kashyap Thimmaraju for helping validate bug reports in an experimental test bed. This work was supported by the following awards and grants: Bundesministerium für Bildung und Forschung (BMBF) under Award No. KIS1DSD032 (Project Enzevalos). The opinions, views, and conclusions contained herein are those of the author(s) and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of BMBF, or any other funding body involved.

References

[1] Clang ast matcher reference. `http://clang.llvm.org/docs/LibASTMatchersReference.html`. Accessed: 31/5/2017.

[2] Clang static analyzer. `https://clang-analyzer.llvm.org/`. Accessed: 25/5/2017.

[3] Clang/llvm sanitizercoverage. `https://clang.llvm.org/docs/SanitizerCoverage.html`. Accessed: 23/5/2017.

[4] CVE-2016-10377. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-10377`. Accessed: 24/5/2017.

[5] CVE-2017-9214. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9214`. Accessed: 24/5/2017.

[6] CVE-2017-9263. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9263`. Accessed: 24/5/2017.

[7] CVE-2017-9264. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9264`. Accessed: 24/5/2017.

[8] CVE-2017-9265. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-9265`. Accessed: 24/5/2017.

[9] gcov: A test coverage program (online documentation). `https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`. Accessed: 25/5/2017.

[10] pinctrl: Be more careful in parsing dhcpv6 and dns. `https://mail.openvswitch.org/pipermail/ovs-dev/2017-May/332712`. Accessed: 24/5/2017.

[11] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataflow tests. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on*, pages 143–151. IEEE, 1995.

[12] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 387–401. IEEE, 2008.

[13] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 368–377. IEEE, 1998.

[14] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9), 2007.

[15] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[16] J. Jang, A. Agrawal, and D. Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 48–62. IEEE, 2012.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[18] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[19] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. In *Reverse engineering*, pages 77–108. Springer, 1996.

[20] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on software Engineering*, 32(3):176–192, 2006.

[21] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 107–114. IEEE, 2001.

[22] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 447–456. ACM, 2010.

[23] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proc. USENIX Annual Technical Conference (ATC)*, pages 28–28, 2012.

[24] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[25] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368. ACM, 2012.

[26] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 797–812. IEEE, 2015.