Under-Constrained Symbolic Execution: Correctness Checking for Real Code

David A. Ramos
ramos@cs.stanford.edu

Dawson Engler
engler@csl.stanford.edu

Stanford University

Abstract

Software bugs are a well-known source of security vulnerabilities. One technique for finding bugs, symbolic execution, considers all possible inputs to a program but suffers from scalability limitations. This paper uses a variant, *under-constrained* symbolic execution, that improves scalability by directly checking individual functions, rather than whole programs. We present UC-KLEE, a novel, scalable framework for checking C/C++ systems code, along with two use cases. First, we use UC-KLEE to check whether patches introduce crashes. We check over 800 patches from BIND and OpenSSL and find 12 bugs, including two OpenSSL denial-of-service vulnerabilities. We also verify (with caveats) that 115 patches do not introduce crashes. Second, we use UC-KLEE as a generalized checking framework and implement checkers to find memory leaks, uninitialized data, and unsafe user input. We evaluate the checkers on over 20,000 functions from BIND, OpenSSL, and the Linux kernel, find 67 bugs, and verify that hundreds of functions are leak free and that thousands of functions do not access uninitialized data.

1 Introduction

Software bugs pervade every level of the modern software stack, degrading both stability and security. Current practice attempts to address this challenge through a variety of techniques, including code reviews, higher-level programming languages, testing, and static analysis. While these practices prevent many bugs from being released to the public, significant gaps remain.

One technique, testing, is a useful sanity check for code correctness, but it typically exercises only a small number of execution paths, each with a single set of input values. Consequently, it misses bugs that are only triggered by other inputs.

Another broad technique, static analysis, is effective at discovering many classes of bugs. However, static analysis generally uses abstraction to improve scalability and cannot reason precisely about program values and pointer relationships. Consequently, static tools often miss deep bugs that depend on specific input values.

One promising technique that addresses the limitations of both testing and static analysis is symbolic execution [4, 5, 40]. A symbolic execution tool conceptually explores all possible execution paths through a program in a bit-precise manner and considers all possible input values. Along each path, the tool determines whether any combination of inputs could cause the program to crash. If so, it reports an error to the developer, along with a concrete set of inputs that will trigger the bug.

Unfortunately, symbolic execution suffers from the well-known *path explosion* problem since the number of distinct execution paths through a program is often exponential in the number of if-statements or, in the worst case, infinite. Consequently, while symbolic execution often examines orders of magnitude more paths than traditional testing, it typically fails to exhaust all interesting paths. In particular, it often fails to reach code deep within a program due to complexities earlier in the program. Even when the tool succeeds in reaching deep code, it considers only the input values satisfying the few paths that manage to reach this code.

An alternative to whole-program symbolic execution is *under-constrained* symbolic execution [18, 42, 43], which directly executes an arbitrary function within the program, effectively *skipping* the costly path prefix from `main` to this function. This approach reduces the number and length of execution paths that must be explored. In addition, it allows library and OS kernel code without a `main` function to be checked easily and thoroughly.

This paper presents UC-KLEE, a scalable framework implementing under-constrained symbolic execution for C/C++ systems code without requiring a manual specification or even a single testcase. We apply this framework to two important use cases. First, we use it to check whether patches to a function introduce new bugs, which may or may not pose security vulnerabilities. Ironically, patches intended to fix bugs or eliminate security vulnerabilities are a frequent source of them. In many cases, UC-KLEE can *verify* (up to a given input bound and with standard caveats) that a patch does not introduce new crashes to a function, a guarantee not possible with existing techniques.

Second, we use UC-KLEE as a general code checking framework upon which specific checkers can be implemented. We describe three example checkers we implemented to find memory leaks, uses of uninitialized data, and unsanitized uses of user input, all of which *may* pose security vulnerabilities. Additional checkers may be added to our framework to detect a wide variety of bugs along symbolic, bit-precise execution paths through functions deep within a program. If UC-KLEE exhaustively checks all execution paths through a function, then it has effectively verified (with caveats) that the function passes the check (e.g., no leaks).

We evaluated these use cases on large, mature, and security-critical code. We validated over 800 patches from BIND [3] and OpenSSL [36] and found 12 bugs, including two OpenSSL denial-of-service vulnerabilities [12, 16]. UC-KLEE verified that 115 patches did not introduce new crashes, and it checked thousands of paths and achieved high coverage even on patches for which it did not exhaust all execution paths.

We applied our three built-in checkers to over 20,000 functions from BIND, OpenSSL, and the Linux kernel and discovered 67 new bugs, several of which appear to be remotely exploitable. Many of these were latent bugs that had been missed by years of debugging effort. UC-KLEE also exhaustively verified (with caveats) that 771 functions from BIND and OpenSSL that allocate heap memory do not cause memory leaks, and that 4,088 functions do not access uninitialized data.

The remainder of this paper is structured as follows: § 2 presents an overview of under-constrained symbolic execution; § 3 and § 4 discuss using UC-KLEE for validating patches and generalized checking, respectively; § 5 describes implementation tricks; § 6 discusses related work; and § 7 concludes.

2 Overview

This paper builds upon our earlier work on UC-KLEE [43], an extension to the KLEE symbolic virtual machine [5] designed to support equivalence verification and under-constrained symbolic inputs. Our tool checks C/C++ code compiled as *bitcode* (intermediate representation) by the LLVM compiler [29]. As in KLEE, it performs bit-accurate symbolic execution of the LLVM bitcode, and it executes any functions called by the code. Unlike KLEE, UC-KLEE begins executing code at an arbitrary function chosen by the user, rather than `main`.

With caveats (described in § 2.2), UC-KLEE provides verification guarantees on a per-path basis. If it exhausts all execution paths, then it has verified that a function has the checked property (e.g. that a patch does not introduce any crashes or that the function does not leak memory) up to the given input size.

Directly invoking functions within a program presents new challenges. Traditional symbolic execution tools generate input values that represent external input sources (e.g., command-line arguments, files, etc.). In most cases, a correct program should reject invalid external inputs rather than crash. By contrast, individual functions typically have *preconditions* imposed on their inputs. For example, a function may require that pointer arguments be non-null. Because UC-KLEE directly executes functions without requiring their preconditions to be specified by the user, the inputs it considers may be a superset (over-approximation) of the legal values handled by the function. Consequently, we denote UC-KLEE's symbolic inputs as *under-constrained* to reflect that they are missing preconditions (constraints).

While this technique allows previously-unreachable code to be deeply checked, the missing preconditions may cause *false positives* (spurious errors) to be reported to the user. UC-KLEE provides both automated heuristics and an interface for users to manually silence these errors by lazily specifying input preconditions using simple C code. In our experience, even simple annotations may silence a large number of spurious errors (see § 3.2.5) and this effort is orders of magnitude less work than eagerly providing a full specification for each function.

2.1 Lazy initialization

UC-KLEE automatically generates a function's symbolic inputs using lazy initialization [26, 46], which avoids the need for users to manually construct inputs, even for complex, pointer-rich data structures. We illustrate lazy initialization by explaining how UC-KLEE executes the example function `listSum` in Figure 1(a), which sums the entries in a linked list. Figure 1(b) summarizes the three execution paths we explore. For clarity, we elide i) error checks that UC-KLEE normally performs at memory accesses, division/remainder operations, and assertions.

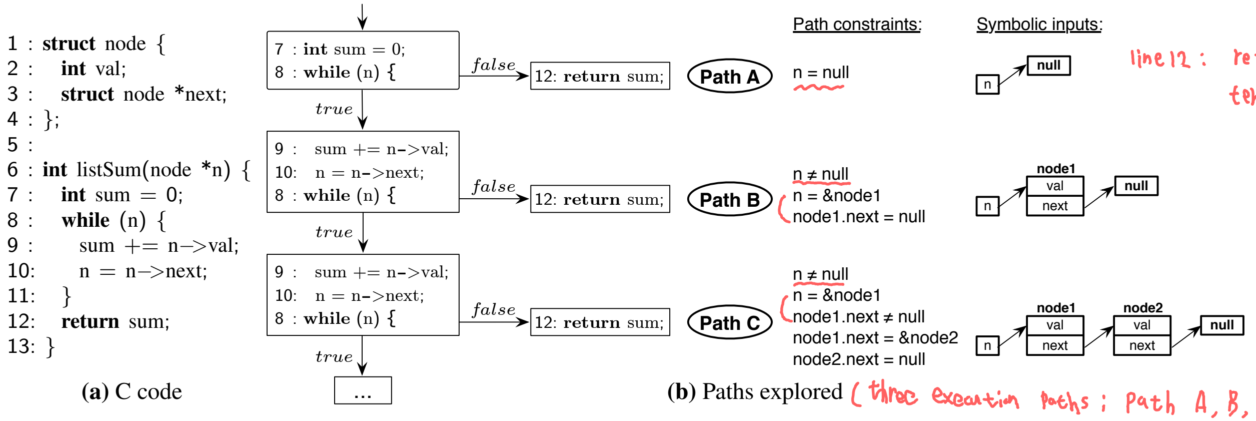
i) omit

UC-KLEE first creates an under-constrained symbolic value to represent the sole argument `n`. Although `n` is a pointer, it begins in the *unbound* state, not yet pointing to any object. UC-KLEE then passes this symbolic argument to `listSum` and executes as follows:

Line 7 The local variable `sum` is assigned a concrete value; no special action is taken.

Line 8 The code checks whether the symbolic variable `n` is non-null. At this point, UC-KLEE forks execution and considers both cases. We first consider the false path where $n = null$, (Path A). We then return to the true path where $n \neq null$ (Path B). On Path A, UC-KLEE adds $n = null$ as a path constraint and skips the loop.

Line 12 Path A returns `0` and terminates.



```
1 : struct node {
2 :   int val;
3 :   struct node *next;
4 : };
5 :
6 : int listSum(node *n) {
7 :   int sum = 0;
8 :   while (n) {
9 :     sum += n->val;
10:     n = n->next;
11:   }
12:   return sum;
13: }
```

(a) C code

(b) Paths explored (three execution paths: Path A, B, and C)

Figure 1: Example code fragment analyzed by UC-KLEE.

line 12: returns 0 and terminates

↳ Goal: automatically synthesize data structures from under-constrained symbolic input

We now consider Path B.

Line 8 UC-KLEE adds the constraint $n \neq null$ and enters the loop.

Line 9 The code dereferences the pointer `n` for the first time on Path B. Because `n` is *unbound*, UC-KLEE allocates a new block of memory, denoted $node1$, to satisfy the dereference and adds the constraint $n = \&node1$ to *bind* the pointer `n` to this object. At this point, `n` is no longer unbound, so subsequent dereferences of that pointer will resolve to $node1$ rather than trigger additional allocations. The (symbolic) contents of $node1$ are marked as unbound, allowing future dereferences of pointers in this object to trigger allocations. This recursive process is the key to lazy initialization. Next, `sum` is incremented by the symbolic value $node1.val$.

Line 10 `n` is set to the value $node1.next$. Path B then returns to the loop header.

Line 8 The code tests whether `n` (set to $node1.next$) is non-null. UC-KLEE forks execution and considers both cases. We first consider $node1.next = null$, which we still refer to as Path B. We will then return to the true path where $node1.next \neq null$ (Path C). On Path B, $node1.next = null$ is added as a path constraint and execution exits the loop.

Line 12 Path B returns $node1.val$ and terminates.

We now consider Path C.

Line 8 UC-KLEE adds $node1.next \neq null$ as a path constraint, and Path C enters the loop.

Line 9 Path C dereferences the unbound symbolic pointer $node1.next$, which triggers allocation of a new object $node2$. This step illustrates the unbounded nature of many loops. To prevent UC-KLEE from allocating an unbounded number of objects as input, the tool accepts a command-line option to limit the depth of an input-derived data structure (k-bounding [17]). When a path attempts to exceed this limit, our tool silently terminates it. For this example, assume a depth limit of two, which causes UC-KLEE to terminate Path D (not shown) at line 9 during the next loop iteration.

Line 10 `n` is set to the value $node2.next$.

Line 8 UC-KLEE forks execution and adds the path constraint $node2.next = null$ to Path C.

Line 12 Path C returns $node1.val + node2.val$ and exits.

*

This example illustrates a simple but powerful recursive technique to automatically synthesize data structures from under-constrained symbolic input. Figure 2 shows an actual data structure our tool generated as input for one of the BIND bugs we discovered (Figure 5). The edges between each object are labeled with the field names contained in the function's debug information and included in UC-KLEE's error report.

2.2 Limitations

Because we build on our earlier version of UC-KLEE, we inherit its limitations [43]. The more important examples are as follows. The tool tests compiled code on a specific platform and does not consider other build configurations. It does not handle assembly (see § 4 for how we skip inline assembly), nor symbolic floating point operations. In addition, there is an explicit assumption that input-derived pointers reference unique objects (no aliasing, and no cyclical data structures), and the tool assigns distinct concrete addresses to allocated objects.

When checking whether patches introduce bugs, UC-KLEE aims to detect crashing bugs and does not look for performance bugs, differences in system call arguments, or concurrency errors. We can only check patches that do not add, remove, or reorder fields in data structures or change the type signatures of patched functions. We plan to extend UC-KLEE to support such patches by implementing a type map that supplies identical inputs to each version of a function in a "field aware" manner. How-

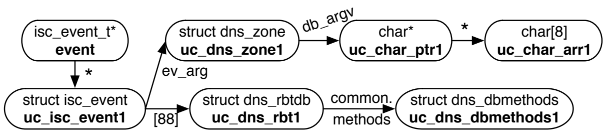


Figure 2: BIND data structure allocated by UC-KLEE.

P : unpatched version
P' : patched version

ever, our current system does not support this, and we excluded such patches from our experiments.

3 Patch checking

To check whether a patch introduces new crashing bugs, UC-KLEE symbolically executes two compiled versions of a function: P, the unpatched version, and P', the patched version. If it finds any execution paths along which P' crashes but P does not (when given the same symbolic inputs), it reports a potential bug in the patch.

Recall that due to missing input preconditions, we cannot simply assume that all crashes are bugs. Instead, UC-KLEE looks for paths that exhibit *differing* crash behavior between P and P', which usually share an identical set of preconditions. Even if UC-KLEE does not know these preconditions, in practice, real code tends to show *error equivalence* [43], meaning that P and P' both crash (or neither crashes) on illegal inputs. For example, if a precondition requires a pointer to be non-null and both versions dereference the pointer, then P and P' will both crash when fed a null pointer as an argument.

In prior work, UC-KLEE [43] verified the equivalence of small library routines, both in terms of crashes and outputs. While detecting differences in functionality may point to interesting bugs, these discrepancies are typically meaningful only to developers of the checked code. Because this paper evaluates our framework on large, complex systems developed by third parties, we limit our discussion to crashes, which objectively point to bugs.

To check patches, UC-KLEE automatically generates a *test harness* that sets up the under-constrained inputs and invokes P and P'. Figure 3 shows a representative test harness. Lines 2–3 create an under-constrained input `n`. Line 4 calls `fooB` (P'). Note that UC-KLEE invokes P' before P to facilitate path pruning (§ 3.1). Line 5 discards any writes performed by `fooB` but preserves the path constraints so that `fooA` (P) will see the same initial memory contents and follow the corresponding path. Line 6 invokes `fooA`.

```
1 : int main() {
2 :     node *n;
3 :     ucklee_make_uc(&n);
4 :     fooB(n); /* run P' */
5 :     ucklee_reset_address_space();
6 :     fooA(n); /* run P */
7 :     return 0;
8 : }
```

Figure 3: Test harness.

If a path through `fooB` crashes, UC-KLEE unwinds the stack and resumes execution at line 5. If `fooA` also crashes on this path, then the two functions are crash equivalent and no error is reported. However, if `fooA` returns from line 6 *without* crashing, we report an error to the user as a possible bug in `fooB`. For this use case, we do not report errors in which `fooA` (P) crashes but `fooB` (P') does not, which suggest bugs *fixed* by a patch.

3.1 Path pruning

UC-KLEE employs several path pruning techniques to target errors and avoid uninteresting paths. The underlying UC-KLEE system includes a *static cross-checker* that walks over the LLVM [29] control flow graph, conservatively marking regions of basic blocks that differ between the original function P and the patched function P'. This algorithm is fairly straightforward, and we elide details for brevity. UC-KLEE soundly prunes paths that:

1. have never executed a "differing" basic block, and
2. cannot reach a differing basic block from their current program counter and call stack.

The second condition uses an inter-procedural reachability analysis from the baseline UC-KLEE system. Paths meeting both of these criteria are safe to prune because they will execute identical instruction sequences.

In addition, UC-KLEE introduces pruning techniques aimed specifically at detecting errors introduced by a patch. As our system executes P' (`fooB` in Figure 3), it prunes paths that either:

1. return from P' without triggering an error, or
2. trigger an error without reaching differing blocks.

In the first case, we are only concerned with errors *introduced* by the patch. In the second case, P and P' would both trigger the error.

Error uniquing. Our system aggressively uniques errors by associating each path executing P with the program counter (PC) of the error that occurred in P'. Once our system executes a non-error path that returns from P (and reports the error in P'), it prunes all current and future paths that hit the same error (PC and type) in P'. In practice, this enabled our system to prune thousands of redundant error paths.

3.2 Evaluation

We evaluated UC-KLEE on hundreds of patches from BIND and OpenSSL, two widely-used, security critical systems. Each codebase contains about 400,000 lines of C code, making them reasonable measures of UC-KLEE's scalability and robustness. For this experiment, we used a maximum symbolic object size of 25,000 bytes and a maximum symbolic data structure depth of 9 objects.

3.2.1 Patch selection and code modifications

We tried to avoid selection bias by using two complete sets of patches from the `git` repositories for recent stable branches: BIND 9.9 from 1/2013 to 3/2014 and OpenSSL 1.0.1 from 1/2012 to 4/2014. Many of the patches we encountered modified more than one function; this section uses *patch* to refer to changes to a single function, and *commit* to refer to a complete changeset.

We excluded all patches that: only changed copyright information, had build errors, modified build infrastructure only, removed dead functions only, applied only to disabled features (e.g., win32), patched only BIND `contrib` features, only touched regression/unit tests, or used variadic functions. We also eliminated all patches

Codebase	Function	Type	Cause	New	Vulnerability
BIND	`receive_secure_db`	assert fail	double lock acquisition	✓	
BIND	`save_nsec3param`	assert fail	uninitialized struct	✓	
BIND	`configure_zone_acl`	assert fail	inconsistent null argument handling	✓	
BIND	`isc_lex_gettoken`	assert fail	input parsing logic	✓	
OpenSSL	`PKCS5_PBKDF2_HMAC`	uninitialized pointer dereference	uninitialized struct		
OpenSSL	`dtls1_process_record`	assert fail	inconsistent null check		
OpenSSL	`tls1_final_finish_mac`	null pointer dereference	unchecked return value	✓	
OpenSSL	`do_ssl3_write`	null pointer dereference	callee side effect after null check	✓	CVE-2014-0198
OpenSSL	`PKCS7_dataDecode`	null pointer dereference	unchecked return value	✓	
OpenSSL	`EVP_DecodeUpdate`	out-of-bounds array access	negative count passed to memcpy	✓	CVE-2015-0292
OpenSSL	`dtls1_buffer_record`	use-after-free	improper error handling	✓	
OpenSSL	`pkey_ctrl_gost`	uninitialized pointer dereference	improper error handling	✓	

Figure 4: Summary of bugs UC-KLEE reported while checking patches. *New* indicates that the bug was previously unknown.

that yielded identical code after compiler optimizations. Because of tool limitations, we excluded patches that changed input datatypes (§ 2.2). Finally, to avoid inflating our verification numbers, we excluded three BIND commits that patched 200-300 functions each by changing a pervasive linked-list macro and/or replacing all uses of `memcpy` with `memmove`. Neither of these changes introduced any errors and, given their near-trivial modifications, shed little additional light on our tool's effectiveness. This yielded 487 patches from BIND and 324 patches from OpenSSL, both from 177 distinct commits to BIND and OpenSSL (purely by coincidence).

We compiled *patched* and *unpatched* versions of the codebase for each revision using an LLVM 2.7 toolchain. We then ran UC-KLEE over each patch for one hour. Each run was allocated a single Intel Xeon E5645 2.4GHz core and 4GB of memory on a compute cluster running 64-bit Fedora Linux 14. For these runs, we configured UC-KLEE to target crashes only in patched routines or routines they call. While this approach allows UC-KLEE to focus on the most likely source of errors, it does not detect bugs caused by the outputs of a function, which may trigger crashes elsewhere in the system (e.g., if the function unexpectedly returns null). UC-KLEE can report such differences, but we elide that feature in this paper.

Code modifications. In BIND and OpenSSL, we canonicalized several macros that introduced spurious code differences such as the `__LINE__`, `VERSION`, `SRCID`, `DATE`, and `OPENSSL_VERSION_NUMBER` macros. To support function-call annotations (§ 3.2.5) in BIND, we converted four preprocessor macros to function calls.

For BIND, we disabled expensive assertion-logging code and much of its debug `malloc` functionality, which UC-KLEE already provided. For OpenSSL, we added a new build target that disabled reference counting and address alignment. The reference counting caused many false positives; UC-KLEE reported double free errors due to unknown preconditions on an object's reference count.

3.2.2 Bugs found (3+8+6 = 17)

From the patches we tested, UC-KLEE uncovered three previously unknown bugs in BIND and eight bugs in OpenSSL, six of which were previously unknown. These bugs are summarized in Figure 4.

```
1 :   LOCK_ZONE(zone);
2 :   if (DNS_ZONE_FLAG(zone, DNS_ZONEFLG_EXITING)
3 :        || !inline_secure(zone)) {
4 :      result = ISC_R_SHUTTINGDOWN;
5 :      goto unlock;
6 :   }
7 :   . . .
8 :   if (result != ISC_R_SUCCESS)
9 :      goto failure; /* ← bypasses UNLOCK_ZONE */
10:   . . .
11: unlock:
12:   UNLOCK_ZONE(zone);
13: failure:
14:   dns_zone_idetach(&zone);
```

Figure 5: BIND locking bug found in `receive_secure_db`.

Figure 5 shows a representative double-lock bug in BIND found by cross-checking. The patch moved the `LOCK_ZONE` earlier in the function (line 1), causing existing error handling code that jumped to `failure` (line 9) to bypass the `UNLOCK_ZONE` (line 12). In this case, the subsequent call to `dns_zone_idetach` (line 14) reacquires the already-held lock, which triggers an assertion failure. This bug was one of several we found that involved infrequently-executed error handling code. Worse, BIND often hides `goto failure` statements inside a `CHECK` macro, which was responsible for a bug we discovered in the `save_nsec3param` function (not shown). We reported the bugs to the BIND developers, who promptly confirmed and fixed them. These examples demonstrate a key benefit of UC-KLEE: it explores non-obvious execution paths that would likely be missed by a human developer, either because the code is obfuscated or an error condition is overlooked.

UC-KLEE is not limited to finding new bugs introduced by the patches; it can also find old bugs in patched code. We added a new mode where UC-KLEE flags errors that occur in both P and P' if the error *must* occur for all input values following that execution path (*must-fail* error described in § 3.2.5). This approach allowed us to find one new bug in BIND and four in OpenSSL. It also re-confirmed a number of bugs found by cross-checking above. This mode could be used to find bugs in functions that have not been patched, but we did not use it for that purpose in this paper.

Figure 6 shows a representative must-fail bug, a previously unknown null pointer dereference (denial-of-service) vulnerability we discovered in OpenSSL's

```
1 : if (wb−>buf == NULL) /* ← null pointer check */
2 :     if (!ssl3_setup_write_buffer(s))
3 :         return −1;
4 : . . .
5 : /* If we have an alert to send, lets send it */
6 : if (s−>s3−>alert_dispatch) {
7 :     /* call sets wb→buf to NULL */
8 :     i=s−>method−>ssl_dispatch_alert(s);
9 :     if (i <= 0)
10:         return(i);
11:     /* if it went, fall through and send more stuff */
12: }
13: . . .
14: unsigned char *p = wb−>buf; /* ← p = NULL */
15: *(p++)=type&0xff; /* ← null pointer dereference */
```

Figure 6: OpenSSL null pointer bug in `do_ssl3_write`.

`do_ssl3_write` function that led to security advisory CVE-2014-0198 [12] being issued. In this case, a developer attempted to prevent this bug by explicitly checking whether `wb->buf` is null (line 1). If the pointer is null, `ssl3_setup_write_buffers` allocates a new buffer (line 2). On line 6, the code then handles any pending alerts [20] by calling `ssl_dispatch_alert` (line 8). This call has the subtle side effect of freeing the write buffer when the common `SSL_MODE_RELEASE_BUFFERS` flag is set. After freeing the buffer, `wb->buf` is set to null (not shown), triggering a null pointer dereference on line 15.

This bug would be hard to find with other approaches. The write buffer is freed by a chain of function calls that includes a recursive call to `do_ssl3_write`, which one maintainer described as "sneaky" [44]. In contrast to static techniques that could not reason precisely about the recursion, UC-KLEE proved that under the circumstances when both an alert is pending and the release flag is set, a null pointer dereference *will* occur. This example also illustrates the weaknesses of regression testing. While a developer may write tests to make sure this function works correctly when an alert is pending *or* when the release flag is set, it is unlikely that a test would exercise these conditions simultaneously. Perhaps as a direct consequence, this vulnerability was nearly six years old.

3.2.3 Patches verified

In addition to finding new bugs, UC-KLEE exhaustively verified all execution paths for 67 (13.8%) of the patches in BIND, and 48 (14.8%) of the patches in OpenSSL. Our system effectively verified that, up to the given input bound and with the usual caveats, these patches did not introduce any new crashes. This strong result is not possible with imprecise static analysis or testing.

The median instruction coverage (§ 3.2.4) for the exhaustively verified patches was 90.6% for BIND and 100% for OpenSSL, suggesting that these patches were thoroughly tested. Only six of the patches in BIND and one in OpenSSL achieved very low (0-2%) coverage. We determined that UC-KLEE achieved low coverage on these patches due to dead code (2 patches); an insufficient symbolic input bound (2 patches); comparisons between input pointers (we assume no aliasing, 1 patch); symbolic `malloc` size (1 patch); and a trivial stub function that was optimized away (1 patch).

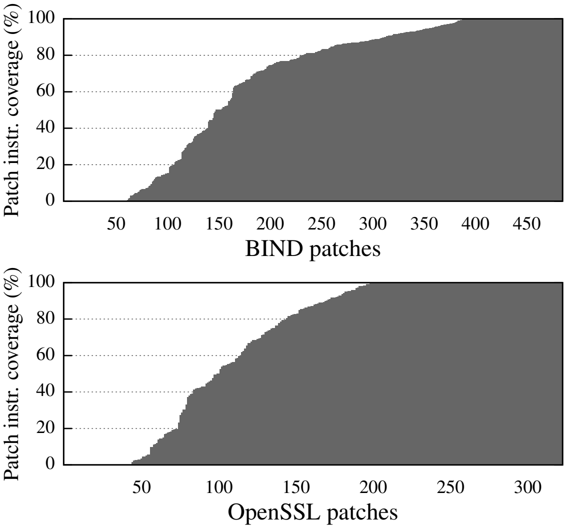


Figure 7: Coverage of patched instructions: 100% coverage for 98 BIND patches (20.1%) and 124 OpenSSL patches (38.3%). Median was 81.1% for BIND, 86.9% for OpenSSL.

3.2.4 Patches partially verified

This section measures how thoroughly we check nonterminating patches using two metrics: (1) instruction coverage, and (2) number of execution paths completed.

We conservatively measure instruction coverage by counting the number of instructions that differ in P' from P and then computing the percentage of these instructions that UC-KLEE executes at least once. Figure 7 plots the instruction coverage. The median coverage was 81.1% for BIND and 86.9% for OpenSSL, suggesting that UC-KLEE thoroughly exercised the patched code, even when it did not exhaust all paths.

Figure 8 plots the number of completed execution paths for each patch we did not exhaustively verify (§ 3.2.3) that hit at least one patched instruction. These graphs exclude 31 patches for BIND and 32 patches for OpenSSL for which our system crashed during the one hour execution window. The crashes were primarily due to bugs in our tool and memory exhaustion/blowup caused by symbolically executing cryptographic ciphers.

For the remaining patches, UC-KLEE completed a median of 5,828 distinct paths per patch for BIND and 1,412 for OpenSSL. At the upper end, 154 patches for BIND (39.6%) and 79 for OpenSSL (32.4%) completed over 10,000 distinct execution paths. At the bottom end, 58 patches for BIND (14.9%) and 46 for OpenSSL (18.9%) completed zero execution paths. In many cases, UC-KLEE achieved high coverage on these patches but neither detected errors nor ran the non-error paths to com-

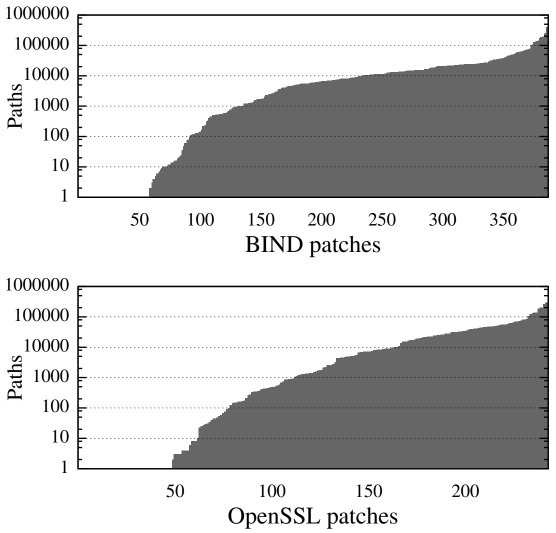


Figure 8: Completed execution paths (log scale). Median was 5,828 paths per patch for BIND and 1,412 for OpenSSL. Top quartile was 17,557 paths for BIND and 21,859 for OpenSSL.

pletion. A few reasons we observed for paths not running to completion included query timeouts, unspecified symbolic function pointers, or ineffective search heuristics.

These numbers should only be viewed as a crude approximation of thoroughness; they do not measure the independence between the paths explored (greater is preferable). On the other hand, they grossly undercount the number of distinct concrete values each symbolic path reasons about simultaneously. One would generally expect that exercising 1,000 or more paths through a patch, where each path simultaneously tests all feasible values, represents a dramatic step beyond the current standard practice of running the patch on a few tests.

3.2.5 False positives

This section describes our experience in separating true bugs from *false positives*, which were due to missing input preconditions. The false positives we encountered were largely due to three types of missing preconditions:

1. Data structure invariants, which apply to all instances of a data structure (e.g., a parent node in a binary search tree has a greater value than its left child).
2. State machine invariants, which determine the sequence of allowed values and the variable assignments that may exist simultaneously (e.g., a counter increases monotonically).
3. API invariants, which determine the legal inputs to API entry points (e.g., a caller must not pass a null pointer as an argument).

Figure 9 illustrates a representative example of a false positive from BIND, which was caused by a missing data structure invariant. The `isc_region_t` type consists of a buffer and a length, but UC-KLEE has no knowledge that the two are related. The code selects a valid buffer length at line 14, the shorter of the two buffers. At line 17, the code calls `memcmp` and supplies this length. Inside `memcmp`, UC-KLEE reported hundreds of false positives involving out-of-bounds memory reads. These errors occurred on false paths where the buffer pointed to by the `base` field was smaller than the associated `length` field.

```
 1 : typedef struct isc_region {
 2 :     unsigned char * base;
 3 :     unsigned int    length;
 4 : } isc_region_t;
 5 :
 6 : int isc_region_compare(isc_region_t *r1, isc_region_t *r2) {
 7 :   unsigned int l;
 8 :   int result;
 9 :
10:    REQUIRE(r1 != NULL);
11:    REQUIRE(r2 != NULL);
12:
13:    /* chooses min. buffer length */
14:    l = (r1->length < r2->length) ? r1->length : r2->length;
15:
16:    /* memcmp reads out-of-bounds */
17:    if ((result = memcmp(r1->base, r2->base, l)) != 0)
18:      return ((result < 0) ? -1 : 1);
19:    else
20:      return ((r1->length == r2->length) ? 0 :
21:        (r1->length < r2->length) ? -1 : 1);
22: }
```

Figure 9: Example false positive in BIND. UC-KLEE does not associate `length` field with buffer pointed to by `base` field. Consequently, UC-KLEE falsely reports that `memcmp` (line 17) reads out-of-bounds from `base`.

UC-KLEE manages false positives using two approaches: manual annotations and automated heuristics.

Manual annotations. UC-KLEE supports two types of manual annotations: (1) data type annotations, and (2) function call annotations. Both are written in C and compiled with LLVM. UC-KLEE invokes data type annotations at the end of a path, prior to emitting an error. These are associated with named data types and specify invariants on symbolic inputs of that type (inferred from debug information when available). For the example above, we added the following simple annotation for the `isc_region_t` data type:

```
INVARIANT(r->length <= OBJECT_SIZE(r->base));
```

The `INVARIANT` macro requires that the condition hold. If it is infeasible (cannot be true) on the current path, UC-KLEE emits an error report with a flag indicating that the annotations have been violated. We use this flag to filter out uninteresting error reports. This one simple annotation allowed us to filter 623 errors, which represented about 7.5% of all the errors UC-KLEE reported for BIND.

Function call annotations are used to run specific code immediately prior to calling a function. For example, we wrote a function call annotation for BIND that runs before each call to `isc_mutex_lock`, with the same arguments:

```
void annot_isc_mutex_lock(isc_mutex_t *mp) {
  EXPECT(*mp == 0);
}
```

	P' only			P and P'		
	Reports		**Patches**	**Reports**		**Patches**
Heuristic	Tot.	Bugs		Tot.	Bugs	
Total errors	2446	3	141	5829	-	260
Manual annotations	1419	3	125	1378	-	153
must-fail	44	3	8	1378	-	153
concrete-fail	26*	2	6*	878	-	110
belief-fail	35*	3	7*	1053	-	127
excluding inputs	30*	3	7*	852	-	102
True bugs	**3***	**3**	**3***	**1**	**1**	**1**

(a) BIND (487 patches, 4 distinct bugs)

	P' only			P and P'		
	Reports		**Patches**	**Reports**		**Patches**
Heuristic	Tot.	Bugs		Tot.	Bugs	
Total errors	1423	5	79	579	11	125
Manual annotations	1286	5	79	451	11	124
must-fail	41	5	22	451	11	124
concrete-fail	14*	5	12*	224	11	98
belief-fail	25*	5	18*	316	11	117
excluding inputs	17*	5	11*	90*	11	47*
True bugs	**5***	**5**	**4***	**11***	**11**	**10***

(b) OpenSSL (324 patches, 8 distinct bugs)

Figure 10: Effects of heuristics on false positives. *Tot.* indicates the total number of reports, of which *Bugs* are true errors; *Patches* indicates the number of patches that reported at least one error. *P' only* refers to errors that occurred only in function P'; *P and P'* occurred in both versions. Indent indicates successive heuristics; * indicates that we reviewed all the reports manually.

Macro	**Description**
`INVARIANT(condition)`	Add `condition` as a path constraint; kill path if infeasible.
`EXPECT(condition)`	Add `condition` as a path constraint if feasible; otherwise, ignore.
`IMPLIES(a, b)`	Logical implication: $a \rightarrow b$.
`HOLDS(a)`	Returns true if condition `a` *must* hold.
`MAY_HOLD(a)`	Returns true if condition `a` *may* hold.
`SINK(e)`	Forces `e` to be evaluated; prevents compiler from optimizing it away.
`VALID_POINTER(ptr)`	Returns true if `ptr` is valid; false otherwise.
`OBJECT_SIZE(ptr)`	Returns the size of the object pointed to by `ptr`; kills path if pointer is invalid.

Figure 11: C annotation macros.

The EXPECT macro adds the specified path constraint only if the condition is feasible on the current path and elides it otherwise. In this example, we avoid considering cases where the mutex is already locked. However, this annotation has no effect if the condition is *not* feasible (i.e., the lock has definitely been acquired along this path). This annotation allows UC-KLEE to detect errors in lock usage while suppressing false positives under the assumption that if a function attempts to acquire a lock supplied as input, then a likely input precondition is that the lock is not already held. This annotation did not prevent us from finding the BIND locking bug in `receive_secure_db` shown in Figure 5.

Figure 11 summarizes the convenience macros we provided for expressing annotations using C code. While annotations may be written using arbitrary C code, these macros provide a simple interface to functionality not expressible with C itself (e.g., determining the size of a heap object using OBJECT_SIZE). The HOLDS and MAY_HOLD macros allow code to check the feasibility of a Boolean expression without causing UC-KLEE to fork execution and trigger path explosion.

For BIND, we wrote 13 function call annotations and 31 data type annotations (about 400 lines of C). For OpenSSL, we wrote six data type annotations and no function call annotations (60 lines). We applied a single set of annotations for each codebase to all the patches we tested. In our experience, most of these annotations were simple to specify and often suppressed many false positives. We felt the level of effort required was reasonable compared to the sizes of the codebases we checked. We added annotations lazily, in response to false positives.

Figure 10 illustrates the effects of the annotations and heuristics on the error reports for BIND and OpenSSL. The *P' only* column describes errors that only occurred in the patched function, while *P and P'* describes errors that occurred in both versions. In this experiment, we are primarily concerned with bugs introduced by a patch, so our discussion describes *P' only* unless otherwise noted.

The manual annotations suppressed 42% of the reports for BIND but only 9.6% for OpenSSL. We attribute this difference to the greater effort we expended writing manual annotations for BIND, for which the automated heuristics were less effective without the annotations.

Automated heuristics. We tried numerous heuristics to reduce false reports. UC-KLEE augments each error report with a list of the heuristics that apply. The **must-fail** heuristic identifies errors that *must* occur for all input values following that execution path, since these are often true errors [18]. For example, assertion failures are must-fail when the condition must be false.

A variation on the must-fail heuristic is the **belief-fail** heuristic, which uses a form of belief analysis [19]. The intuition behind this heuristic is that if a function *contradicts* itself, it likely has a bug. For example, if the code checks that a pointer is null and then dereferences the pointer, it has a bug, regardless of any input preconditions. On the other hand, a function is generally agnostic to the assumptions made by the functions it calls. For example, if `strcmp` checks whether two strings have the same address, the caller does not acquire this belief, even if the path constraints now indicate that the two addresses match. Following this intuition, the belief-fail heuristic identifies errors that occur for all input values satisfying the *belief set*, which is the set of constraints (i.e., branch conditions) added within the current function or inherited from its caller, but not its callees. We track belief sets for each stack frame.

A second variation on must-fail is **concrete-fail**, which indicates that an assertion failure or memory error was triggered by a concrete (non-symbolic) condition or pointer, respectively. In practice, this heuristic and belief-fail were the most effective.

These heuristics reduced the total number of reports to a small enough number that we were able to inspect them all manually. While only 8.6% of the belief-fail errors for BIND and 20% of those for OpenSSL were true bugs, the total number of these errors (60) was manageable relative to the number of patches we tested (811). In total, the annotations and belief-fail heuristic eliminated 98.6% of false positives for BIND and 98.2% for OpenSSL.

A subset of the belief-fail errors were caused by reading past the end of an input buffer, and none of these were true bugs. Instead, they were due to paths reaching the input bound we specified. In many cases, our system would emit these errors for any input bound because they involved unbounded loops (e.g., `strlen`). The *excluding inputs* row in Figure 10 describes the subset of belief-fail errors not related to input buffers. This additional filter produced a small enough set of *P and P′* errors for OpenSSL that we were able to manually inspect them, discovering a number of additional bugs. We note that the *true errors* listed in Figure 10 constitute 12 distinct bugs; some bugs showed up in multiple error reports.

4 Generalized checking

In addition to checking patches, UC-KLEE provides an interface for rule-based checkers to be invoked during symbolic path exploration. These checkers are similar to tools built using dynamic instrumentation systems such as Valgrind [34] or Pin [30]. Unlike these frameworks, however, UC-KLEE applies its checkers to all possible paths through a function, not to a single execution path through a program. In addition, UC-KLEE considers all possible input values along each path, allowing it to discover bugs that might be missed when checking a single set of concrete inputs.

Conceptually, our framework is similar to WOODPECKER [8], a KLEE-based tool that allows system-specific checkers to run on top of (whole program) symbolic execution. In this paper, however, we focus on generic checkers we implemented for rules that apply to many systems, and we directly invoked these checkers on individual functions deep within each codebase.

UC-KLEE provides a simple interface for implementing checkers by deriving from a provided C++ base class. This interface provides hooks for a checker to intercept memory accesses, arithmetic operations, branches, and several types of errors UC-KLEE detects.

A user invoking UC-KLEE provides a compiled LLVM module and the name of a function to check. We refer to this function as the *top-level* function. Generally, the module has been linked to include all functions that might be called by the top-level function. When UC-KLEE encounters a function call, it executes the called function. When UC-KLEE encounters a call to a function missing from the LLVM module, however, it may optionally skip over the function call rather than terminate the path with an error message. When UC-KLEE skips a function call, it creates a new under-constrained value to represent the function's return value, but it leaves the function's arguments unchanged. This approach *under-approximates* the behaviors that the missing function might perform (e.g., writing to its arguments or globals). Consequently, UC-KLEE may miss bugs and cannot provide verification guarantees when functions are missing.

We briefly experimented with an alternative approach in which we overwrote the skipped function's arguments with new under-constrained values, but this over-approximation caused significant path explosion, mostly involving paths that could not arise in practice.

In addition to missing functions due to scalability limitations, we also encountered inline assembly (Linux kernel only) and unresolved symbolic function pointers. We skipped these two cases in the same manner as missing functions. For all three cases, UC-KLEE provides a hook to allow a checker to detect when a call is being skipped and to take appropriate actions for that checker.

In the remainder of this section, we describe each checker, followed by our experimental results in § 4.4.

4.1 Leak checker

Memory leaks can lead to memory exhaustion and pose a serious problem for long-running servers. Frequently, they are exploitable as denial-of-service vulnerabilities [10, 13, 14]. To detect memory leaks (which may or may not be remotely exploitable, depending on their location within a program), we implemented a leak checker on top of UC-KLEE. The leak checker considers a heap object to be leaked if, after returning from the top-level function, the object is not reachable from a *root set* of pointers. The root set consists of a function's (symbolic) arguments, its return value, and all global variables. This checker is similar to the leak detection in Purify [23] or Valgrind's memcheck [34] tool, but it thoroughly checks all paths through a specific function, rather than a single concrete path through a whole program.

When UC-KLEE encounters a missing function, the leak checker finds the set of heap objects that are reachable from each of the function call's arguments using a precise approach based on pointer referents [42, 43]. It then marks these objects as *possibly escaping*, since the missing function could capture pointers to these objects and prevent them from becoming unreachable. At the end of each execution path, the leak checker removes any possibly escaping objects from the set of leaked objects.

Doing so allows it to report only true memory leaks, at the cost of possibly omitting leaks when functions are missing. However, UC-KLEE may still report false leaks along invalid execution paths due to missing input preconditions. Consider the following code fragment:

```
1 : char* leaker() {
2 :     char *a = (char*) malloc(10); /* not leaked */
3 :     char *b = (char*) malloc(10); /* maybe leaked */
4 :     char *c = (char*) malloc(10); /* leaked! */
5 :
6 :     bar(b); /* skipped call to bar */
7 :     return a;
8 : }
```

When UC-KLEE returns from the function `leaker`, it inspects the heap and finds three allocated objects: `a`, `b`, and `c`. It then examines the root set of objects. In this example, there are no global variables and `leaker` has no arguments, so the root set consists only of `leaker`'s return value. UC-KLEE examines this return value and finds that the pointer `a` is live (and therefore not leaked). However, neither `b` nor `c` is reachable. It then looks at its list of *possibly escaping* pointers due to the skipped call to `bar` on line 6, which includes `b`. UC-KLEE subtracts `b` from the set of leaked objects and reports back to the user that `c` has been leaked. While this example is trivial, UC-KLEE discovered 37 non-trivial memory leak bugs in BIND, OpenSSL, and the Linux kernel (§ 4.4).

4.2 Uninitialized data checker

Functions that access uninitialized data from the stack or heap exhibit undefined or non-deterministic behavior and are particularly difficult to debug. Additionally, the prior contents of the stack or heap may hold sensitive information, so code that operates on these values may be vulnerable to a loss of confidentiality.

UC-KLEE includes a checker that detects accesses to uninitialized data. When a function allocates stack or heap memory, the checker fills it with special *garbage* values. The checker then intercepts all loads, binary operations, branches, and pointer dereferences to check whether any of the operands (or the result of a load) contain garbage values. If so, it reports an error to the user.

In practice, loads of uninitialized data are often intentional; they frequently arise within calls to `memcpy` or when code manipulates bit fields within a C `struct`. Our evaluation in § 4.4 therefore focuses on branches and dereferences of uninitialized pointers.

When a call to a missing function is skipped, the uninitialized data checker *sanitizes* the function's arguments to avoid reporting spurious errors in cases where missing functions write to their arguments.

4.3 User input checker

Code that handles untrusted user input is particularly prone to bugs that lead to security vulnerabilities since an attacker can supply any possible input value to exploit the code. Generally, UC-KLEE treats inputs to a function as *under-constrained* because they may have unknown preconditions. For cases where inputs originate from untrusted sources such as network packets or user-space data passed to the kernel, however, the inputs can be considered *fully-constrained*. This term indicates that the set of legal input values is known to UC-KLEE; in this case, any possible input value may be supplied. If any value triggers an error in the code, then the error is likely to be exploitable by an attacker, assuming that the execution path is feasible (does not violate other preconditions).

UC-KLEE maintains *shadow memory* (metadata) associated with each symbolic input that tracks whether each symbolic byte is under-constrained or fully-constrained. UC-KLEE provides an interface for system-specific C annotations to mark untrusted inputs as fully-constrained by calling the function `ucklee_clear_uc_byte`. This function sets the shadow memory for each byte to the *fully-constrained* state.

UC-KLEE includes a system-configurable user input checker that intercepts all errors and adds an `UNSAFE_INPUT` flag to errors caused by fully-constrained inputs. For memory access errors, the checker examines the pointer to see if it contains fully-constrained symbolic values. For assertion failures, it examines the assertion condition. For division-by-zero errors, it examines the divisor.

In all cases, the checker inspects the fully-constrained inputs responsible for an error and determines whether any path constraints compare the inputs to under-constrained data (originating elsewhere in the program). If so, the checker assumes that the constraints *may* properly sanitize the input, and it suppresses the error. Otherwise, it emits the error. This approach avoids reporting spurious errors to the user, at the cost of missing errors when inputs are partially (but insufficiently) sanitized.

We designed this checker primarily to find security vulnerabilities similar to the OpenSSL "Heartbleed" vulnerability [1, 11] from 2014, which passed an untrusted and unsanitized length argument to `memcpy`, triggering a severe loss of confidentiality. In that case, the code never attempted to sanitize the length argument. To test this checker, we ran UC-KLEE on an old version of OpenSSL without the fix for this bug and confirmed that our checker reports the error.

4.4 Evaluation

We evaluated UC-KLEE's checkers on over 20,000 functions from BIND, OpenSSL, and the Linux kernel. For BIND and OpenSSL, we used UC-KLEE to check all functions except those in the codebases' `test` directories. We used the same minor code modifications described in § 3.2.1, and we again used a maximum input

	Leak Checker				Uninitialized Data Checker					User Input Checker			
	Funcs.	Bugs	Reports	False	Funcs.	Bugs	Pointer Reports	Pointer False	Branch Reports	Funcs.	Bugs	Reports	False
BIND	6239	9	138	2.2%	6239	3	0	-	244*	6239	0	67	100%
OpenSSL	6579	5	272†	90.1%	6579	6	197	92.90%	564*	6579	0	5	100%
Linux kernel	5812	23	127	76.4%	7185	10	72	83.30%	494*	1857	11	145	80.0%

Figure 12: Summary of results from running UC-KLEE checkers on *Funcs* functions from each codebase. *Bugs* shows the number of distinct true bugs found (67 total). *Reports* shows the total number of errors reported by UC-KLEE in each category (multiple errors may point to a single bug). *False* reports the percentage of errors reported that did not appear to be true bugs (i.e., false positives). †excludes reports for obfuscated ASN.1 code. *denotes that we inspected only a handful of errors for that category.

```
1 : int gssp_accept_sec_context_upcall(struct net *net,
2 :                                     struct gssp_upcall_data *data) {
3 :   . . .
4 :   ret = gssp_alloc_receive_pages(&arg);
5 :   . . .
6 :   gssp_free_receive_pages(&arg);
7 :   . . .
8 : }
9 : int gssp_alloc_receive_pages(struct gssx_arg_accept_sec_context *arg) {
10:   arg−>pages = kzalloc(. . .);
11:   . . .
12:   return 0;
13: }
14: void gssp_free_receive_pages(struct gssx_arg_accept_sec_context *arg) {
15:   for (i = 0; i < arg−>npages && arg−>pages[i]; i++)
16:     __free_page(arg−>pages[i]);
17:   /* missing: kfree(arg−>pages); */
18: }
```

Figure 13: Linux kernel memory leak in RPCSEC_GSS protocol implementation used by NFS server-side AUTH_GSS.

size of 25,000 bytes and a depth bound of 9 objects.

For the Linux kernel, we included functions relevant to each checker, as described below. Unlike our evaluation in § 3.2, we did not use any manual annotations to suppress false positives. We ran UC-KLEE for up to five minutes on each function from BIND and the Linux kernel, and up to ten minutes on each OpenSSL function. We used the same machines as in § 3.2.

For BIND, we checked version 9.10.1-P1 (12/2014). For OpenSSL, we checked version 1.0.2 (1/2015). For the Linux kernel, we checked version 3.16.3 (9/2014).

Figure 12 summarizes the results. UC-KLEE discovered a total of 67 previously-unknown bugs[1]: 12 in BIND, 11 in OpenSSL, and 44 in the Linux kernel. Figure 14 lists the number of functions that UC-KLEE exhaustively verified (up to the given input bound and with caveats) as having each property. We omit verification results from the Linux kernel because UC-KLEE skipped many function calls and inline assembly, causing it to under-approximate the set of possible execution paths and preventing it from making any verification guarantees. We did link each Linux kernel function with other modules from the same directory, however, as well as the `mm/vmalloc.c` module.

	No leaks	No `malloc`	No uninitialized data
BIND	388	1776	2045
OpenSSL	383	1648	2043

Figure 14: Functions verified (with caveats) by UC-KLEE.

4.4.1 Leak checker

The leak checker was the most effective. It reported the greatest number of bugs (37 total) and the lowest false positive rate. Interestingly, only three of the 138 leak reports for BIND were spurious errors, a false positive rate of only 2.2%. For OpenSSL, we excluded 269 additional reports involving the library's obfuscated ASN.1 [25] parsing code, which we could not understand. Of the remaining 272 reports, the checker found five bugs but had a high false positive rate of 90.1%.

For the Linux kernel, we wrote simple C annotations (about 60 lines) to intercept calls to `kmalloc`, `vmalloc`, `kfree`, `vfree`, and several similar functions, and to forward these to UC-KLEE's built-in `malloc` and `free` functions. Doing so allowed us to track memory management without the overhead of symbolically executing the kernel's internal allocators. We then ran UC-KLEE on all functions that directly call these allocation functions.

Our system discovered 23 memory leaks in the Linux kernel. One particularly interesting example (Figure 13) involved the SunRPC layer's server-side implementation of `AUTH_GSS` authentication for NFS. Each connection triggering an upcall causes 512 bytes allocated at line 10 to be leaked due to a missing `kfree` that should be present around line 17. Since this leak may be triggered by remote connections, it poses a potential denial-of-service (memory exhaustion) vulnerability. The NFS maintainers accepted our patch to fix the bug.

UC-KLEE found that at least 2909 functions in BIND and at least 3700 functions in OpenSSL (or functions they call) allocate heap memory. As shown in Figure 14, UC-KLEE verified (with caveats) that 388 functions in BIND and 383 in OpenSSL allocate heap memory but do not leak it. Our system also verified that 1776 functions in BIND and 1648 functions in OpenSSL do not allocate heap memory, making them trivially leak-free.

4.4.2 Uninitialized data checker

The uninitialized data checker reported a total of 19 new bugs. One illustrative example, shown in Figure 15, in-

[1] A complete list of the bugs we discovered is available at: `http://cs.stanford.edu/~daramos/usenix-sec-2015`

```
1 :   points = OPENSSL_malloc(sizeof (EC_POINT*)*(num + 1));
2 :   ...
3 :   for (i = 0; i < num; i++) {
4 :     if ((points[i] = EC_POINT_new(group)) == NULL)
5 :       goto err; /* leaves ’points’ only partially initialized */
6 :   }
7 :   ...
8 : err:
9 :   ...
10:   if (points) {
11:     EC_POINT **p;
12:     for (p = points; *p != NULL; p++)
13:       EC_POINT_free(*p); /* dereference/free of uninitialized pointer */
14:     OPENSSL_free(points);
15:   }
```

Figure 15: OpenSSL dereference/`free` of uninitialized pointer in `ec_wNAF_precompute_mult` function.

volves OpenSSL’s elliptic curve cryptography. If the call to `EC_POINT_new` on line 4 fails, the code jumps to line 8, leaving the `points` array partially uninitialized. Line 13 then passes uninitialized pointers from the array to `EC_POINT_free`, which dereferences the pointers and passes them to `free`, potentially corrupting the heap. This is one of many bugs that we found involving infrequently executed error-handling code, a common source of security bugs.

UC-KLEE discovered an interesting bug (Figure 16) in BIND’s UDP port randomization fix for Kaminsky’s cache poisoning attack [9]. To prevent spoofed DNS replies, BIND must use unpredictable source port numbers. The `dispatch_createudp` function calls the `get_udpsocket` function at line 9, which selects a pseudorandom number generator (PRNG) at line 18 based on whether we are using a UDP or TCP connection. However, the `socktype` field isn’t initialized in `dispatch_createudp` until line 12, meaning that the PRNG selection is based on uninitialized data. While it appears that the resulting port numbers are sufficiently unpredictable despite this bug, this example illustrates UC-KLEE’s ability to find errors with potentially serious security implications.

For the Linux kernel, we checked the union of the functions we used for the leak checker and the user input checker (discussed below) and found 10 bugs.

Due to time limitations, we exhaustively inspected only the most serious category of errors: uninitialized pointers. The checker reported too many uninitialized branches for us to examine completely, but we did inspect a few dozen of these errors in an ad-hoc manner. All three of the bugs from BIND and one bug from the Linux kernel fell into this category. The remaining bugs were uninitialized pointer errors. We did not inspect the error reports for binary operations or load values.

Finally, our system verified (with caveats) that about a third of the functions from BIND (2045) and OpenSSL (2043) do not access uninitialized data. We believe that providing this level of guarantee on such a high percentage of functions with almost no manual effort is a strong result not possible with existing tools.

```
1 : #define DISP_ARC4CTX(disp) \
2 :   ((disp)−>socktype == isc_sockettype_udp) \
3 :   ? (&(disp)−>arc4ctx) : (&(disp)−>mgr−>arc4ctx)
4 : static isc_result_t dispatch_createudp(..., unsigned int attributes, ...) {
5 :   ...
6 :   result = dispatch_allocate(mgr, maxrequests, &disp);
7 :   ...
8 :   if ((attributes & DNS_DISPATCHATTR_EXCLUSIVE) == 0) {
9 :     result = get_udpsocket(mgr, disp, ...);
10:     ...
11:   }
12:   disp−>socktype = isc_sockettype_udp; /* late initialization */
13:   ...
14: }
15: static isc_result_t get_udpsocket(..., dns_dispatch_t *disp, ...) {
16:   ...
17:   /* PRNG selected based on uninitialized ’socktype’ field */
18:   prt = ports[dispatch_uniformrandom(DISP_ARC4CTX(disp), nports)];
19:   ...
20: }
```

Figure 16: BIND non-deterministic PRNG selection bug.

4.4.3 User input checker

The user input checker required us to identify data originating from untrusted sources. Chou [6] observed that data swapped from network byte order to host byte order is generally untrusted. We applied this observation to OpenSSL and used simple annotations (about 40 lines of C) to intercept calls to `n2s`, `n2l`, `n2l3`, `n2l6`, `c2l`, and `c2ln`, and mark the results fully-symbolic. We also applied a simple patch to OpenSSL to replace byte-swapping macros with function calls so that UC-KLEE could use our annotations. We hope to explore automated ways of identifying untrusted data in future work.

For BIND, we annotated (about 50 lines) the byte-swapping functions `ntohs` and `ntohl`, along with `isc_buffer_getuint8` and three other functions that generally read from untrusted buffers.

For the Linux kernel, we found that many network protocols store internal state in network byte order, leading to spurious errors if we consider these to be untrusted. Instead, we annotated (about 40 lines) the `copy_from_user` function and `get_user` macro (which we converted to a function call). In addition, we used an option in UC-KLEE to mark all arguments to the system call handlers `sys_*` as untrusted. Finally, we used UC-KLEE to check the 1502 functions that directly invoke `copy_from_user` and `get_user`, along with the 355 system call handlers in our build.

Reassuringly, this checker did not discover any bugs in the latest versions of BIND or OpenSSL. We attribute this both to the limited amount of data we marked as untrusted and to our policy of suppressing errors involving *possibly* sanitized data (see § 4.3). However, we were able to detect the 2014 “Heartbleed” vulnerability [1, 11] when we ran our system on an old version of OpenSSL.

Interestingly, we did discover 11 new bugs in the Linux kernel. Seven of these bugs were division- or

```
1 : static int dg_dispatch_as_host(..., struct vmci_datagram *dg) {
2 :    /* read length field from userspace datagram */
3 :    dg_size = VMCI_DG_SIZE(dg);
4 :    ...
5 :    dg_info = kmalloc(sizeof(*dg_info) +
6 :            (size_t) dg->payload_size, GFP_ATOMIC);
7 :    ...
8 :    /* unchecked memcpy length; read overrun */
9 :    memcpy(&dg_info->msg, dg, dg_size);
10:    ...
11: }
```

Figure 17: Linux kernel VMware Communication Interface driver unchecked `memcpy` length (buffer overread) bug.

remainder-by-zero operations that would trigger floating-point exceptions and crash the kernel. The remaining four bugs are out-of-bounds dereferences.

Figure 17 shows a buffer overread bug we discovered in the kernel driver for the VMware Communication Interface (VMCI) that follows a pattern nearly identical to "Heartbleed." The userspace datagram `dg` is read using `copy_from_user`. The code then allocates a destination buffer on line 5 and invokes `memcpy` on line 9 without sanitizing the `dg_size` field read from the datagram. An attacker could potentially use this bug to copy up to 69,632 bytes of private kernel heap memory and send it from the host OS to the guest OS. Fortunately, this vulnerability is only exploitable by code running locally on the host OS. The maintainers quickly patched this bug.

Figure 18 shows an unsanitized remainder-by-zero bug we found in the kernel driver for the CEPH distributed filesystem. The check at line 6 aims to prevent this bug with a 64-bit comparison, but the divisor at line 8 uses only the low 32 bits of the untrusted `stripe_unit` field (read from userspace using `copy_from_user`). A value such as `0xffffffff00000000` would pass the check but result in a remainder-by-zero error. An unprivileged local attacker could potentially issue an `ioctl` system call to crash the machine. We notified the developers, who promptly fixed the bug.

Because of the ad-hoc nature of this checker, we did not use it to exhaustively verify any properties about the functions we checked.

5 Implementation

This section details optimizations and techniques we implemented to scale our framework and address problems we encountered while applying it to large systems.

5.1 Object sizing

Recall that when an unbound symbolic pointer is dereferenced, UC-KLEE must allocate memory and bind the pointer to it. One challenge in implementing this functionality is picking a useful object size to allocate. If the size is too small, later accesses to this object may trigger out-of-bounds memory errors. On the other hand, a size that is too large can hide legitimate errors. We handled this tradeoff using two approaches.

```
1 : static long _validate_layout(..., struct ceph_ioctl_layout *l) {
2 :    ...
3 :    /* validate striping parameters */
4 :    if ((l->object_size & ~PAGE_MASK) ||
5 :         (l->stripe_unit & ~PAGE_MASK) ||
6 :         (l->stripe_unit != 0 && /* ← 64-bit check */
7 :                                    /* 32-bit divisor: */
8 :          ((unsigned)l->object_size % (unsigned)l->stripe_unit)))
9 :            return -EINVAL;
10:    ...
11: }
```

Figure 18: Linux kernel CEPH distributed filesystem driver remainder-by-zero bug in `ioctl` handler.

The first approach, which we used for our experiment in § 3.2, implemented a form of *backtracking*. At each unbound pointer dereference, UC-KLEE *checkpoints* the execution state and chooses an initial allocation size using a heuristic that examines any available type information [42]. If the path later reads out-of-bounds from this object, UC-KLEE (1) emits the error to the user, and (2) restores the checkpoint and uses an allocation size large enough to satisfy the most recent memory access. UC-KLEE records the sequence of branches taken after each checkpoint, and it forces the path to *replay* the sequence of branches after increasing the allocation size. In practice, replaying branches exposed many sources of nondeterminism in the baseline KLEE tool and its system modeling code, which we were able to eliminate through significant development effort.

An alternative approach that we recently incorporated into UC-KLEE is to use symbolically-sized objects, rather than selecting a single concrete size. Doing so avoids the need for backtracking in most cases by simultaneously considering many possible object sizes. At each memory access, UC-KLEE determines whether the offset could exceed the object's symbolic size. If so, it emits an error to the user. It also considers a path on which the offset does not exceed this bound and adds a path constraint that sets a lower bound on the object's size. We used this approach for our evaluation in § 4.4.

5.2 Error reporting

With whole program symbolic execution, symbolic inputs typically represent unstructured strings or byte arrays from command line arguments or file contents. In this case, an error report typically contains a single set of concrete inputs that trigger the error, along with a backtrace. With under-constrained symbolic execution, however, the inputs are often complex, pointer-rich data structures since UC-KLEE directly executes individual functions within a program. In this case, a single set of concrete values is not easily understood by a user, nor can it be used to trivially reproduce the error outside of UC-KLEE because pointer inputs expect memory objects (i.e., stack, heap, and globals) to be located at specific addresses.

To provide more comprehensible error reports, UC-

KLEE emits a *path summary* for each error. The path summary provides a complete listing of the source code executed along the path, along with the path constraints added by each line of source. The path constraints are expressed in a C-like notation and use the available LLVM debug information to determine the types and names of each field. Below we list example constraints that UC-KLEE included with error reports for BIND (§ 3.2):

Code:	`REQUIRE(VALID_RBTDB(rbtdb));`
Constraint:	`uc_dns_rbtdb1.common.impmagic == 1380074548`
Code:	`if (source->is_file)`
Constraint:	`uc_inputsource1.is_file == 0`
Code:	`if (c == EOF)`
Constraint:	`uc_var2[uc_var1.current + 1] == 255`

5.3 General KLEE optimizations

We added several scalability improvements to UC-KLEE that apply more broadly to symbolic execution tools. To reduce path explosion in library functions such as `strlen`, we implemented special versions that avoid forking paths by using symbolic if-then-else constructs. We also introduced scores of rules to simplify symbolic expressions [42]. We elide further details due to space.

5.3.1 Lazy constraints

During our experiments, we faced query timeouts and low coverage for several benchmarks that we traced to symbolic division and remainder operations. The worst cases occurred when an unsigned remainder operation had a symbolic value in the denominator. To address this challenge, we implemented a solution we refer to as *lazy constraints*. Here, we defer evaluation of expensive queries until we find an error. In the common case where an error does not occur or two functions exhibit crash equivalence along a path, our tool avoids ever issuing potentially expensive queries. When an error is detected, the tool re-checks that the error path is feasible (otherwise the error is invalid).

Figure 19(a) shows a simple example. With *eager constraints* (the standard approach), the if-statement at line 2 triggers an SMT query involving the symbolic integer division operation `y / z`. This query may be expensive, depending on the other path constraints imposed on `y` and `z`. To avoid a potential query timeout, UC-KLEE introduces a lazy constraint (Figure 19(b)). On line 1, it replaces the result of the integer division operation with a new, unconstrained symbolic value `lazy_x` and adds the lazy constraint `lazy_x = y / z` to the current path. At line 2, the resulting SMT query is the trivial expression `lazy_x > 10`. Because `lazy_x` is unconstrained, UC-KLEE will take both the *true* and *false* branches following the if-statement. One of these branches may violate the constraints imposed on `y` and `z`, so UC-KLEE must check that the lazy constraints are consistent with the full set of path constraints prior to emitting any errors to the user (i.e., if the path later crashes).

```
1 : int x = y / z;
2 : if (x > 10) /* query: y / z > 10 */
3 :   ...
```

(a) Eager constraints (standard)

```
1 : int x = lazy_x; /* adds lazy constraint: lazy_x = y / z */
2 : if (x > 10) /* query: lazy_x > 10 */
3 :   ...
```

(b) Lazy constraints

Figure 19: Lazy constraint used for integer division operation.

In many cases, the delayed queries are more efficient than their eager counterparts because additional path constraints added after the division operation have narrowed the solution space considered by the SMT solver. If our tool determines that the path is infeasible, it silently terminates the path. Otherwise, it reports the error to the user.

5.4 Function pointers

Systems such as the Linux kernel, BIND, and OpenSSL frequently use function pointers within `struct` types to emulate object-oriented methods. For example, different function addresses may be assigned depending on the version negotiated for an SSL/TLS connection [20]. This design poses a challenge for our technique because symbolic inputs contain symbolic function pointers. When our tool encounters an indirect call through one of these pointers, it is unclear how to proceed.

We currently require that users specify concrete function pointers to associate with each type of object (as the need arises). When our tool encounters an indirect call through a symbolic pointer, it looks at the object's debug type information. If the user has defined function pointers for that type of object, our tool executes the specified function. Otherwise, it reports an error to the user and terminates the path. The user can leverage these errors to specify function pointers only when necessary. For BIND, we found that most of these errors could be eliminated by specifying function pointers for only six types: three for memory allocation, and three for internal databases. For OpenSSL, we specified function pointers for only three objects: two related to support for multiple SSL/TLS versions, and one related to I/O.

When running UC-KLEE's checkers, we optionally allow the tool to skip unresolved function pointers, which allows it to check more code but prevents verification guarantees for the affected functions (see § 4).

6 Related work

This paper builds on prior work in symbolic execution [4], particularly KLEE [5] and our early work on UC-KLEE [43]. Unlike our previous work, which targeted small library routines, this paper targets large systems and supports generalized checking.

Other recent work has used symbolic execution to check patches. DiSE [39] performs whole program symbolic execution but prunes paths unaffected by a patch. Differential Symbolic Execution (DSE) [38] and regression verification [21] use abstraction to achieve scalability but may report false differences. By contrast, our approach soundly executes complete paths through each patched function, eliminating this source of false positives. Impact Summaries [2] complement our approach by soundly pruning paths and ignoring constraints unaffected by a patch.

SymDiff [27] provides a scalable solution to check the equivalence of two programs with fixed loop unrolling but relies on imprecise, uninterpreted functions. Differential assertion checking (DAC) [28] is the closest to our work and applies SymDiff to the problem of detecting whether properties that hold in P also hold in P', a generalization of crash equivalence. However, DAC suffers from the imprecisions of SymDiff and reports false differences when function calls are reordered by a patch. Abstract semantic differencing [37] achieves scalability through clever abstraction but, as with SymDiff, suffers additional false positives due to over-approximation.

Recent work has used symbolic execution to generate regression tests exercising the code changed by a patch [41, 31, 32]. While they can achieve high coverage, these approaches use existing regression tests as a starting point and greedily redirect symbolic branch decisions toward a patch, exploring only a small set of execution paths. By contrast, our technique considers all possible intermediate program values as input (with caveats).

Dynamic instrumentation frameworks such as Valgrind [34] and PIN [30] provide a flexible interface for checkers to examine a program's execution at runtime and flag errors. However, these tools instrument a single execution path running with concrete inputs, making them only as effective as the test that supplies the inputs.

Similar to our use of generalized checking in UC-KLEE is WOODPECKER [8], which uses symbolic execution to check system-specific rules. Unlike UC-KLEE, WOODPECKER applies to whole programs, so we expect it would not scale well to large systems. However, WOODPECKER aggressively prunes execution paths that are redundant with respect to individual checkers, a technique that would be useful in UC-KLEE.

Prior work in memory leak detection has used static analysis [45], dynamic profiling [24], and binary rewriting [23]. Dynamic tools such as Purify [23] and Valgrind [34] detect a variety of memory errors at runtime, including uses of uninitialized data. CCured [33] uses a combination of static analysis and runtime checks to detect pointer errors. Our user input checker relates to prior work in dynamic taint analysis, including TaintCheck [35] and Dytan [7].

7 Conclusions and future work

We have presented UC-KLEE, a novel framework for validating patches and applying checkers to individual C/C++ functions using under-constrained symbolic execution. We evaluated our tool on large-scale systems code from BIND, OpenSSL, and the Linux kernel, and we found a total of 79 bugs, including two OpenSSL denial-of-service vulnerabilities.

One avenue for future work is to employ UC-KLEE as a tool for finding general bugs (e.g., out-of-bounds memory accesses) in a single version of a function, rather than cross-checking two functions or using specialized checkers. Our preliminary experiments have shown that this use case results in a much higher rate of false positives, but we did find a number of interesting bugs, including the OpenSSL denial-of-service attack for which advisory CVE-2015-0291 [15, 22, 42] was issued.

In addition, we hope to further mitigate false positives by using ranking schemes to prioritize error reports, and by inferring invariants to reduce the need for manual annotations. In fact, many of the missing input preconditions can be thought of as consequences of a weak type system in C. We may target higher-level languages in the future, allowing our framework to assume many built-in invariants (e.g., that a length field corresponds to the size of an associated buffer).

Acknowledgements

The authors would like to thank Joseph Greathouse and the anonymous reviewers for their valuable feedback. In addition, the authors thank the LibreSSL developers for their quick responses to our bug reports, along with Evan Hunt and Sue Graves of ISC for granting us access to the BIND `git` repository before it became public. This work was supported by DARPA under agreements 1190029-276707 and N660011024088, by the United States Air Force Research Laboratory (AFRL) through contract FA8650-10-C-7024, and by a National Science Foundation Graduate Research Fellowship under grant number DGE-0645962. The views expressed in this paper are the authors' own.

References

[1] Alert (TA14-098A): OpenSSL 'Heartbleed' vulnerability (CVE-2014-0160). `https://www.us-cert.gov/ncas/alerts/TA14-098A`, April 2014.

[2] BACKES, J., PERSON, S., RUNGTA, N., AND TKACHUK, O. Regression verification using impact summaries. In *Proc. of SPIN Symposium on Model Checking of Software (SPIN)* (2013).

[3] BIND. `https://www.isc.org/downloads/bind/`.

[4] BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. Select – a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices 10*, 6 (June 1975), 234–45.

[5] CADAR, C., DUNBAR, D., AND ENGLER, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex sys-

tems programs. In *Proc. of Symp. on Operating Systems Design and Impl (OSDI)* (2008).

[6] CHOU, A. On detecting heartbleed with static analysis. `http://security.coverity.com/blog/2014/Apr/on-detecting-heartbleed-with-static-analysis.html`, 2014.

[7] CLAUSE, J., LI, W., AND ORSO, A. Dytan: a generic dynamic taint analysis framework. In *Proc. of Intl. Symp. on Software Testing and Analysis (ISSTA)* (2007).

[8] CUI, H., HU, G., WU, J., AND YANG, J. Verifying systems rules using rule-directed symbolic execution. In *Proc. of Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2013).

[9] CVE-2008-1447: DNS Cache Poisoning Issue (”Kaminsky bug”). `https://kb.isc.org/article/AA-00924`.

[10] CVE-2012-3868. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-3868`, Jul 2012.

[11] CVE-2014-0160. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160`, April 2014.

[12] CVE-2014-0198. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0198`, May 2014.

[13] CVE-2014-3513. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3513`, Oct 2014.

[14] CVE-2015-0206. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0206`, Jan 2015.

[15] CVE-2015-0291. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0291`, Mar 2015.

[16] CVE-2015-0292. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0292`, Mar 2015.

[17] DENG, X., LEE, J., AND ROBBY. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *Proc. of the 21st IEEE International Conference on Automated Software Engineering* (2006), pp. 157–166.

[18] ENGLER, D., AND DUNBAR, D. Under-constrained execution: making automatic code destruction easy and scalable. In *Proc. of the Intl. Symposium on Software Testing and Analysis (ISSTA)* (2007).

[19] ENGLER, D., YU CHEN, D., HALLEM, S., CHOU, A., AND CHELF, B. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (2001).

[20] FREIER, A. *RFC 6101: The Secure Sockets Layer (SSL) Protocol Version 3.0.* Internet Engineering Task Force (IETF), Aug 2011.

[21] GODLIN, B., AND STRICHMAN, O. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability 23*, 3 (2013), 241–258.

[22] GOODIN, D. OpenSSL warns of two high-severity bugs, but no Heartbleed. *Ars Technica* (March 2015).

[23] HASTINGS, R., AND JOYCE, B. Purify: Fast detection of memory leaks and access errors. In *Proc. of the USENIX Winter Technical Conference (USENIX Winter '92)* (Dec. 1992), pp. 125–138.

[24] HAUSWIRTH, M., AND CHILIMBI, T. M. Low-overhead memory leak detection using adaptive statistical profiling. In *Proc. of the Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2004).

[25] INTERNATIONAL TELECOMMUNICATION UNION. *ITU-T Recommendation X.680: Abstract Syntax Notation One (ASN.1): Specification of basic notation*, Nov 2008.

[26] KHURSHID, S., PASAREANU, C. S., AND VISSER, W. Generalized symbolic execution for model checking and testing. In *Proc. of Intl. Conf. on Tools and Algos. for the Construction and Analysis of Sys.* (2003).

[27] LAHIRI, S., HAWBLITZEL, C., KAWAGUCHI, M., AND REBELO, H. SymDiff: A language-agnostic semantic diff tool for imperative programs. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)* (2012).

[28] LAHIRI, S. K., MCMILLAN, K. L., SHARMA, R., AND HAWBLITZEL, C. Differential assertion checking. In *Proc. of Joint Meeting on Foundations of Software Engineering (FSE)* (2013).

[29] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. of the Intl. Symp. on Code Generation and Optimization (CGO)* (2004).

[30] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (2005).

[31] MARINESCU, P. D., AND CADAR, C. High-coverage symbolic patch testing. In *Proc. of Intl. SPIN Symp. on Model Checking Software* (2012).

[32] MARINESCU, P. D., AND CADAR, C. KATCH: High-coverage testing of software patches. In *Proc. of 9th Joint Mtg. on Foundations of Software Engineering (FSE)* (2013).

[33] NECULA, G. C., MCPEAK, S., AND WEIMER, W. Ccured: type-safe retrofitting of legacy code. In *Proc. of Symp. on Principles of Programming Languages (POPL)* (2002).

[34] NETHERCOTE, N., AND SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)* (June 2007), pp. 89–100.

[35] NEWSOME, J., AND SONG, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of Network and Distributed Systems Security Symp. (NDSS)* (2005).

[36] OpenSSL. `https://www.openssl.org/source`.

[37] PARTUSH, N., AND YAHAV, E. Abstract semantic differencing for numerical programs. In *Proc. of Intl. Static Analysis Symposium (SAS)* (2013).

[38] PERSON, S., DWYER, M. B., ELBAUM, S., AND PĂSĂREANU, C. S. Differential symbolic execution. In *Proc. of ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering (FSE)* (2008), pp. 226–237.

[39] PERSON, S., YANG, G., RUNGTA, N., AND KHURSHID, S. Directed incremental symbolic execution. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)* (2011).

[40] PĂSĂREANU, C. S., AND RUNGTA, N. Symbolic PathFinder: Symbolic execution of java bytecode. In *Proc. of the IEEE/ACM International Conf. on Automated Software Engineering (ASE)* (2010).

[41] QI, D., ROYCHOUDHURY, A., AND LIANG, Z. Test generation to expose changes in evolving programs. In *Proc. of IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)* (2010).

[42] RAMOS, D. A. *Under-constrained symbolic execution: correctness checking for real code*. PhD thesis, Stanford University, 2015.

[43] RAMOS, D. A., AND ENGLER, D. R. Practical, low-effort equivalence verification of real code. In *Proc. of Intl. Conf. on Computer Aided Verification (CAV)* (2011).

[44] UNANGST, T. Commit e76e308f (tedu): on today's episode of things you didn't want to learn. `http://anoncvs.estpak.ee/cgi-bin/cgit/openbsd-src/commit/lib/libssl?id=e76e308f`, Apr 2014.

[45] XIE, Y., AND AIKEN, A. Context- and path-sensitive memory leak detection. In *Proc. of the Intl. Symp. on Foundations of Software Engineering (FSE)* (2005).

[46] XIE, Y., AND AIKEN, A. Scalable error detection using boolean satisfiability. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)* (2005), pp. 351–363.