Chopped Symbolic Execution

David Trabish
Tel Aviv University
Israel
davivtra@post.tau.ac.il

Andrea Mattavelli
Imperial College London
United Kingdom
amattave@imperial.ac.uk

Noam Rinetzky
Tel Aviv University
Israel
maon@cs.tau.ac.il

Cristian Cadar
Imperial College London
United Kingdom
c.cadar@imperial.ac.uk

ABSTRACT

Symbolic execution is a powerful program analysis technique that systematically explores multiple program paths. However, despite important technical advances, symbolic execution often struggles to reach deep parts of the code due to the well-known path explosion problem and constraint solving limitations.

In this paper, we propose *chopped symbolic execution*, a novel form of symbolic execution that allows users to specify uninteresting parts of the code to exclude during the analysis, thus only targeting the exploration to paths of importance. However, the excluded parts are not summarily ignored, as this may lead to both false positives and false negatives. Instead, they are executed lazily, when their effect may be observable by code under analysis. Chopped symbolic execution leverages various on-demand static analyses at runtime to automatically exclude code fragments while resolving their side effects, thus avoiding expensive manual annotations and imprecision.

Our preliminary results show that the approach can effectively improve the effectiveness of symbolic execution in several different scenarios, including failure reproduction and test suite augmentation.

CCS CONCEPTS

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

KEYWORDS

Symbolic execution, Static analysis, Program slicing

ACM Reference Format:
David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. 2018. Chopped Symbolic Execution. In *ICSE '18: 40th International Conference on Software Engineering, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3180155.3180251

1 INTRODUCTION

Symbolic execution lies at the core of many modern techniques to software testing, automatic program repair, and reverse engineering [3, 11, 16, 24, 32, 35]. At a high-level, symbolic execution systematically explores multiple paths in a program by running the code with symbolic values instead of concrete ones. Symbolic execution engines thus replace concrete program operations with ones that manipulate symbols, and add appropriate constraints on the symbolic values. In particular, whenever the symbolic executor reaches a branch condition that depends on the symbolic inputs, it determines the feasibility of both sides of the branch, and creates two new independent *symbolic states* which are added to a worklist to follow each feasible side separately. This process, referred to as *forking*, refines the conditions on the symbolic values by adding appropriate constraints on each path according to the conditions on the branch. Test cases are generated by finding concrete values for the symbolic inputs that satisfy the *path conditions*. To both determine the feasibility of path conditions and generate concrete solutions that satisfies them, symbolic execution engines employ *satisfiability-modulo theory* (SMT) constraint solvers [19].

The Challenge. Symbolic execution has proven to be effective at finding subtle bugs in a variety of software [3, 11, 12, 25, 39], and has started to see industrial take-up [13, 15, 25]. However, a key remaining challenge is scalability, particularly related to constraint solving cost and path explosion [14].

Symbolic execution engines issue a huge number of queries to the constraint solver that are often large and complex when analyzing real-world programs. As a result, constraint solving dominates runtime for the majority of non-trivial programs [30, 33]. Recent research has tackled the challenge by proposing several constraint solving optimizations that can help reduce constraint solving cost [5, 12, 21, 27, 33–35, 41, 45].

Path explosion represents the other big challenge facing symbolic execution, and the main focus of this paper. Path explosion refers to the challenge of navigating the huge number of paths in real programs, which is usually at least exponential to the number of static branches in the code. The common mechanism employed by symbolic executors to deal with this problem is the use of search heuristics to prioritise path exploration. One particularly effective heuristic focuses on achieving high coverage by guiding the exploration towards the path closest to uncovered instructions [10–12, 43]. In practice, these heuristics only partially alleviate the path explosion problem, as the following example demonstrates.

Motivating Example. The `extract_octet()` function, shown in Figure 1, is a simplified version of a function from the `libtasn1` library which parses ASN.1 encoding rules from an input string.[1] The ASN.1 protocol is used in many networking and cryptographic applications, such as those handling public key certificates and electronic mail. Versions of `libtasn1` before 4.5 are affected by a heap-overflow security vulnerability that could be exploited via a crafted certificate.[2] Unfortunately, given a time budget of 24 hours,

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

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180251

[1] https://www.gnu.org/software/libtasn1
[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-3622

```
1 int extract_octet(asn_t asn, char *str, int str_len) {
2   int len3, counter, counter_end, result;
3   int len2 = get_length(str, str_len, &len3);
4   counter = len3+1;
5   counter_end = str_len;
6   while (counter < counter_end) {
7     // call to get_length() leads to a heap overflow:
8     len2 = get_length(str+counter, str_len, &len3);
9     if (len2 >= 0) {
10      DECR_LEN(str_len, len2+len3);
11      append_value(asn, str+counter+len3, len2);
12    } else {
13      DECR_LEN(str_len, len3);
14      result = extract_octet(asn, str+counter+len3, str_len);
15      if (result != SUCCESS)
16        return result;
17      len2 = 0;
18    }
19    // str_len should have been decremented at the
20    // beginning of the while block
21    DECR_LEN(str_len, 1);
22    counter += len2+len3+1;
23  }
24  return SUCCESS;
25 }
```

Figure 1: A simplified excerpt from the `extract_octet` routine in `libtasn1`. The invocation of `get_length()` in line 8 leads to a heap overflow because `str_len` has not been decremented before the call.

the analysis of the `extract_octet()` function using the state-of-the-art symbolic execution engine KLEE [11] fails to identify the vulnerability due to path explosion.

At each loop iteration (lines 6–23), the function decodes the length of the current data element with `get_length` (line 8). Function `get_length` scans through the input string and decodes the ASN.1 fields. Then, the execution either recursively iterates over the input string (line 14), or invokes `append_value` (line 11). Function `append_value` creates the actual node in the Abstract Syntax Tree (AST) by decoding the input string given the obtained length. This function scans once more over the input string, performs several checks over the selected element, and allocates memory for the node in the recursive data structure.

Path explosion in this function occurs due to several nested function calls. Symbolically executing function `get_length` alone with a symbolic string of n characters leads to $4 * n$ different paths. Function `append_value` increases even more the number of paths and also affects the efficiency of the symbolic execution engine due to a huge number of constraint solver invocations. As a result, the symbolic executor fails to identify the heap-overflow vulnerability at line 8.

Our Approach. Identifying the vulnerability from the entry point of the library is not trivial: To reach the faulty invocation of function `get_length`, the input triggering the vulnerability traverses 2,945 calls to 98 different functions, for a total amount of 386,727 instructions. Our key observation is that most of the functions required during the execution are *not relevant* for finding the vulnerability. The vulnerability occurs due to an incorrect update of the remaining bytes for parsing (line 21), which results in a memory out-of-bound read when calling `get_length`. The bug thus occurs in code which deals with the parsing, which is independent from the functions that construct the corresponding ASN.1 representation, such as `append_value`. Therefore, we could have quickly reached the bug if we had *skipped* the irrelevant functions that build the AST.

In this paper, we propose a novel form of symbolic execution called *chopped symbolic execution* that provides the ability to specify parts of the code to exclude during the analysis, thus enabling symbolic execution to focus on significant paths only. The skipped code is not trivially excluded from symbolic execution, since this may lead to spurious results. Instead, chopped symbolic execution lazily executes the relevant parts of the excluded code when explicitly required. In this way, chopped symbolic execution does not sacrifice the soundness guarantees provided by standard symbolic execution—except for non-termination of the skipped functions, which may be considered a bug on its own—in that only feasible paths are explored, but effectively discards paths irrelevant to the task at hand.

We developed a prototype implementation of chopped symbolic execution and report the results of an initial experimental evaluation that demonstrates that this technique can indeed lead to efficient and effective exploration of the code under analysis.

Main Contributions. In summary, in this paper we make the following contributions:

(1) We introduce *chopped symbolic execution*, a novel form of symbolic execution that leverages a lightweight specification of uninteresting code parts to significantly improve the scalability of symbolic execution, without sacrificing soundness.
(2) We present Chopper, a prototype implementation of our technique within KLEE [11], and make it publicly available.
(3) We report on an experimental evaluation of Chopper in two different scenarios: failure reproduction and test suite augmentation, and show that chopped symbolic execution can improve and respectively complement standard symbolic execution.

This paper is organised as follows. Section 2 gives a high-level overview of chopped symbolic execution, and Section 3 presents our technique in detail. Section 4 briefly discusses our implementation inside the KLEE symbolic execution engine. Section 5 presents the experimental evaluation of our technique, and in particular it shows that chopped symbolic execution can overcome the limitations of state-of-the-art symbolic executors. Section 6 surveys the main approaches related to this work. Section 7 summarises the contributions of the paper and describes ongoing research work.

2 OVERVIEW

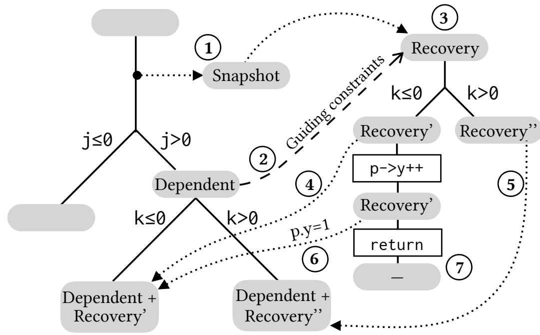
In this section, we give a high-level overview of chopped symbolic execution using the simple program in Figure 2. In particular, Figure 2a shows the entry point of the program (function `main`), while Figure 2c shows the uninteresting code which we would like to skip (function `f`).

We start the chopped execution by executing `main` symbolically. When a state reaches the function call for `f` at line 7, we create a *snapshot state* by cloning the current state, and skip the function call. The snapshot state is shown graphically in Figure 2b, where each gray oval represents a symbolic execution state.

With a snapshot created, we then continue the execution on the current state, but from this point we must consider that some

```
1  struct point { int x, y, z;};
2
3  int main() {
4      struct point p = {0, 0, 0};
5      int j, k; // symbolic
6
7      f(&p, k); // skip
8      if (j > 0)
9         if (p.y)
10           bug();
11     else
12        allgood();
13     return 0;
14 }
```

(a)



(b)

```
15 void f(struct point *p, int k) {
16     if (k % 2)
17        p->z++;
18
19     if (k > 0)
20        p->x++;
21     else
22        p->y++;
23 }
```

(c)

Figure 2: Graphical illustration of chopped symbolic execution on a simple example.

load instructions may depend on the *side effects* of the skipped function f, i.e. the memory locations that f may update. In our example, the side effects of f are the memory locations pointed to by p.z, p.x, and p.y, which are updated at lines 17, 20, and 22 respectively. (We compute the side effects of f using conservative static *pointer analysis* [4, 26, 37] before the symbolic exploration starts, see §3.) We define those instructions that read from the side effects of the skipped functions as *dependent loads*.

On some paths, symbolic execution does not encounter such dependent loads. For example, the path following the else side of the branch at line 8 accesses neither p.x nor p.y nor p.z, so no further action is needed on those paths, and the exploration may correctly terminate without ever going through the code of f. Indeed, in real programs there are often paths that do not depend on the skipped functions, and in such cases symbolic execution immediately benefits from our approach: irrelevant paths are safely skipped, thus reducing path explosion.

However, on other paths symbolic execution encounters *dependent loads*. This happens for our example on the path which explores the then side of the branch at line 8, when it loads the value of p.y at line 9. At this point, the current state needs to be suspended until the relevant paths in function f are explored, and becomes a *dependent state*. To recover a path, we create a new *recovery state* which inherits the snapshot state generated before skipping f at line 7 and start executing symbolically the function.

While symbolic execution is in the recovery state, if the execution forks, then the same fork is performed in the dependent state. Furthermore, as we run the recovery state, any stores to the memory location read by the dependent load are also performed in the dependent state. For example, if the symbolic execution of f traverses the else branch at lines 21–22, then the value of p.y (the memory location pointed to by p->y) is set to 1 in the dependent state too. If the recovery state returns successfully, the dependent state is resumed successfully. If an error occurs while executing the recovery state (e.g., an invalid memory access or a division by zero error, which could have occurred if p->z were set in line 17 to 4/p->y) the corresponding dependent state is terminated.

When we execute a recovery state, not all paths might be compatible with the execution which the dependent state reached. For example, if line 8 were changed from if (j>0) to if (k>0), then the dependent state would have $k > 0$ in its path condition, rendering the dependent state incompatible with the path in f where $k \leq 0$.

One way to filter such incompatible paths would be to execute all possible paths thorough f during recovery, and later filter the ones that are incompatible with the dependent state. However, this would potentially lead to the exploration of a large number of infeasible paths. We thus designed a more efficient solution: Each state maintains a list of *guiding constraints*, which are those constraints added since the call to the skipped function. In our example, the guiding constraints for the dependent state are $j > 0$. Before we execute a recovery state, we add these *guiding constraints* from the dependent state to the path condition of the recovery state. By doing this, we guarantee that every path explored in the recovery state is consistent with respect to its dependent state.

During recovery, one could execute all possible paths through the skipped function f which are compatible with the dependent state, as we could in the example above. However, for real programs this could be unnecessarily expensive, as many paths do not influence the dependent load which started the recovery. To avoid this possible path explosion, and reduce the cost of constraint solving, we aim to only execute the paths that could influence the dependent load. We accomplish this by statically *slicing* [7, 40, 42, 44] the function f with respect to the store instructions that write to the memory location read by the dependent load, that is, the side effects observable from the dependent load. Note that function f could call other functions, so the slicing is done for all these functions too. In our example, the slicing would likely be able to completely remove the if statement at lines 16–17, which would halve the number of explored paths, thus reducing path explosion. It would also likely remove the then side of the if statement at line 19, which in this case does not bring significant benefits, but it could, if that side of the branch were replaced by say, an expensive loop. Slicing away these code parts is possible because they do not update p.y on which the dependent load on line 9 relies.[3]

[3]In practice, the success of the slicing algorithm in reducing the size of the code depends on the precision of the underlying pointer analysis.

Figure 2b shows how chopped symbolic execution works on our example in a graphical way. To recapitulate, when the call to `f` is reached at line 7, a snapshot state is created by cloning the current state (step ① in the figure). Then, on the execution state that reaches line 9, the current state becomes a dependent state and is suspended (step ②), and a recovery state is created by cloning the snapshot state and adding the guiding constraints from the dependent state (step ③). At this point, function f is statically sliced with respect to the dependent load, in our case removing the first *if* statement and the *then* slide of the second *if* statement. Then, the recovery state starts symbolically running the sliced version of `f`. When execution is forked at line 19, then the dependent state is also forked along the same constraints (steps ④ and ⑤). One of the forked recovery states (Recovery') updates the location `p->y` on which our dependent load relies on, so this location is also updated in the corresponding dependent state (step ⑥). Finally, when a recovery state terminates, it gets discarded (step ⑦), and symbolic execution is resumed from its dependent states and other normal states in the program.

3 CHOPPED SYMBOLIC EXECUTION

In this section, we describe our technique in detail and provide the background regarding the main static analysis it employs, namely *pointer analysis* [4, 26, 37].

Algorithm 1 presents the key steps in chopped symbolic execution, which we gradually explain. The algorithm operates on a simple imperative C-like heap-manipulating language with assignments, assertions, conditional jumps, dynamic memory allocation and reclamation, and function calls with call-by-value parameter passing.[4] Functions may have pointer parameters. Thus, without loss of generality, we assume that functions do not have a return value.[5] To simplify the explanation, we now assume that we may skip at most one function invocation at every explored path, and discuss the general case in §3.3.[6] For the same reason, we also assume that the program does not dynamically allocate memory, and discuss this aspect in §3.4.

Chopped symbolic execution begins by invoking function CSE with an *initial symbolic state* (s_0) and a set containing the names of the functions that the user wishes to skip (*skipFunctions*). We expect a symbolic state s to encode, among other properties, the next instruction to be executed (denoted by NEXTINSTRUCTION(s)), the activation record stack, and a (symbolic) description of the program *heap*. For example, the chopped symbolic execution described in Section 2 begins with s_0 in which the stack contains only the activation record of `main`, with the next instruction at line 4, an empty heap, and *skipFunctions* = {`f`}.

At the beginning of the algorithm the *worklist* is empty (line 1), and we initialize it with s_0 (line 3). Then, a standard worklist-based algorithm starts executing until either the worklist is empty (line 4), or the algorithm exhausts the time budget (elided). As usual, the algorithm selects a symbolic state s to explore out of the worklist (line 5). Unconventionally, however, the worklist only has the states which are not suspended, as suspended states are blocked until the

Algorithm 1 Chopped symbolic execution (simplified).

```
1:  worklist ← ∅
2:  function CSE(s₀, skipFunctions)
3:    worklist ← worklist ∪ {s₀}
4:    while worklist ≠ ∅ do
5:      s ← SELECT(worklist)
6:      inst ← NEXTINSTRUCTION(s)
7:      switch inst do
8:        case Call
9:          f ← TARGETFUNCTION(s)
10:         if f ∈ skipFunctions then
11:           snapshot ← CREATESNAPSHOT(s)
12:           s.skipped ← s.skipped + (f, snapshot)
13:         else
14:           EXECUTECALL(s)
15:       case Load
16:         addr ← GETLOADADDRESS(s)
17:         if MAYMOD(s, s.skipped, addr) then
18:           CREATERECOVERYSTATE(s, addr)
19:         else
20:           EXECUTELOAD(s, inst)
21:       case Branch
22:         if s.isRecoveryState then
23:           dependentState ← GETDEPENDENT(s)
24:           φ = CONDITION(inst)
25:           s′ ← FORK(s, φ)
26:           dependentState′ ← FORK(dependentState, φ)
27:           if FEASIBLE(s′) ∧ FEASIBLE(dependentState′) then
28:             worklist ← worklist ∪ {s′}
29:           s′′ ← FORK(s, ¬φ)
30:           dependentState′′ ← FORK(dependentState, ¬φ)
31:           if FEASIBLE(s′′) ∧ FEASIBLE(dependentState′′) then
32:             worklist ← worklist ∪ {s′′}
33:           worklist ← worklist \ {s}
34:         else
35:           EXECUTEBRANCH(s)
36:       case Store
37:         addr ← GETSTOREADDRESS(s)
38:         EXECUTESTORE(s, addr)
39:         if s.isRecoveryState then
40:           UPDATEDEPENDENTSTATE(s, addr)
41:         else
42:           s.overwrittenSet ← s.overwrittenSet ∪ {addr}
43:       case Return
44:         if s.isRecoveryState ∧ RETURNINSKIPPED(s) then
45:           TERMINATE(recoveryState)
46:           dependentState ← GETDEPENDENT(s)
47:           RESUME(dependentState)
48:           worklist ← worklist ∪ {dependentState}
49:         else
50:           EXECUTERETURN(s)
51:     end switch
```

[4] Our implementation operates on LLVM bitcode [29].

[5] A function with a return value can always be rewritten with an additional parameter that points to the memory location of the return value.

[6] For completeness of presentation, Algorithms 1, 2 and 3 handle the general case.

Algorithm 2 Auxiliary function: createRecoveryState

```
1:  function createRecoveryState(dependentState, addr)
2:    foreach (f, snapshot) ∈ dependentState.skipped do
3:      if mayMod(dependentState, (f, snapshot), addr) then
4:        suspend(dependentState)
5:        gc ← getGuidingConstraints(dependentState)
6:        recoveryState ← fork(snapshot, gc)
7:        recoveryState.isRecoveryState ← true
8:        slice(recoveryState, addr)
9:        linkDependent(recoveryState, addr, dependentState)
10:       worklist ← worklist ∪ {recoveryState}
```

Algorithm 3 Auxiliary function: mayMod

```
1:  function mayMod(s, funclist, addr)
2:    foreach (f, snapshot) ∈ funclist do
3:      if allocSite(s, addr) ∈ modSet(f) then
4:        if addr ∉ s.overwrittenSet then
5:          return true
6:    return false
```

value of the depended load is resolved (see §2). The next step of the algorithm depends on the instruction type (line 7).

Handling Call **instructions** (lines 8–14): A Call instruction is handled as illustrated by step ① in Figure 2 (see §2): First, the algorithm determines the name f of the invoked function (line 9). Then, if f is one of the skipped functions, the algorithm creates a snapshot of the current state s (line 11) and records the *snapshot* state at the end of its list of *skipped invocations* (line 12). A skipped invocation is represented as a tuple (f, *snapshot*) composed of the name of a skipped function f, and a *snapshot* of the symbolic state at the time f was skipped.

Conversely, if f should not be skipped, the algorithm handles its invocation as usual in symbolic execution. For brevity, we omit the standard handling of commands by symbolic execution.

Handling Load **instructions** (lines 15–20): Chopped symbolic execution uses mayMod(s, s.*skipped*, *addr*), shown in Algorithm 3 and explained in §3.1, to determine whether the address from which a value is read (*addr*) might have been modified by one of the skipped functions on the path followed by the current state s. If so, the algorithm generates recovery states by calling createRecovery–State(s, *addr*). Otherwise, the Load instruction is handled as usual in symbolic execution (line 20).

Function createRecoveryState is shown in Algorithm 2. The function handles Load instructions as illustrated by step ② in Figure 2 (see §2): It iterates over the list of skipped functions (line 2), and uses mayMod to determine which of the skipped functions f might have modified *addr* (line 3). Once it finds such a function, the current dependent state is immediately suspended (line 4). The function then generates the corresponding recovery state *recoveryState* by forking *snapshot* and by augmenting its path condition with the guiding constraints *gc* (lines 5 to 7), i.e. the path constraints accumulated in s since the snapshot state was created. The algorithm then invokes a static program slicer to remove from the skipped function f instructions which cannot affect the address of the dependent load (line 8); records that *dependentState* spawned *recoveryState* to determine the value written in address *addr* of *dependentState* (line 9); and pushes the recovery state into the worklist (line 10).

Handling Branch **instructions** (lines 21–35): The algorithm checks whether the current state s is a recovery state. If so, then the Branch instruction is handled as illustrated by steps ④ and ⑤ in Figure 2 (see §2): It first retrieves the (suspended) dependent state *dependentState*, which spawned s as a recovery state (line 23). It then determines the branch condition φ (line 24); forks both the current (recovery) state s and the dependent state *dependentState*, and adds φ to their path condition (lines 25–26). After the fork, it checks whether the resulting states are feasible, i.e. their path conditions are satisfiable (line 27), and if so, adds the new recovery state to the worklist (line 28). If either one is not feasible, the newly forked recovery and dependent states are simultaneously discarded. Lines 29–32 act similarly to lines 25–28, except that we use the negation of the branch condition $\neg\varphi$. Finally, the original recovery state s is removed from the worklist (line 33). If the state s is not a recovery state, then the Branch instruction is handled as usual in symbolic execution (line 35).

Handling Store **instructions** (lines 36–42): The algorithm executes the Store instruction on the current state in two steps. First, it performs the actual store (lines 37–38). If s is a recovery state, then the algorithm invokes updateDependentState (line 40, function body elided for space reasons) to update the dependent state, as illustrated by step ⑥ in Figure 2. Otherwise, if s is not a recovery state, it updates the set of overwritten addresses in the current state to record that a value was stored in *addr* after the skipped invocation, and thus any value they may write is no longer relevant (line 42).

Handling Return **instructions** (lines 43–50): If s is a recovery state *and* the Return instruction is invoked inside the skipped function (line 44), then the recovery is terminated and the instruction is handled as illustrated by step ⑦ in Figure 2 (see §2): Specifically, the recovery state itself is discarded (line 45) and the dependent state is resumed (lines 46–47). Otherwise, the Return instruction is handled as usual in symbolic execution (line 50).

3.1 Static Inference of Function Side-Effects

The auxiliary function mayMod(s, *funclist*, *addr*), shown in Algorithm 3, receives as parameters a symbolic state s, a list of skipped invocations *funclist*, and an address *addr* which is the target of a Load instruction, and determines whether one of the skipped functions in *funclist* may store a value in *addr*. The function makes this decision using a *points-to* graph computed by a preliminary pointer analysis stage [26, 37].

More specifically, we perform a whole-program flow-insensitive, context-insensitive, and field-sensitive points-to analysis which determines, in a conservative way, the memory location each pointer variable may point to. In this analysis, memory locations are conservatively abstracted using their *allocation sites*: Every definition of a local or a global variable is considered to be an allocation site, as well as every program point in which memory is allocated. For example, if the program contains `while (..) do L: p=malloc(4)` then we represent all the memory locations allocated in L by a single allocation site AS_L. We then say that `p` may point to allocation site AS_L, and if the program contains `p=q`, we say the same about

q. The nodes of the points-to graph of a program are the variable names and allocation sites, and its edges represent *points-to relations*: An edge from node v to w means that the memory location represented by v may hold a pointer to w.

The points-to graph, which is computed once for every program, conservatively represents all the possible points-to relations in any possible program execution. Using the points-to graph, we use a standard *may-mod* analysis (see, e.g., [1]), in which we find the side effects of every function f, i.e. the set of possible locations, represented by their allocation sites, that the function itself or any function that it may (transitively) invoke, may modify.

During the chopped symbolic execution, we instrument the symbolic state to record the allocation site of every memory location. This instrumentation, together with the program points-to graph, allows mayMod to determine whether a skipped function may write to a given address. Recall that the pointer analysis is flow-insensitive, and thus it might record that a skipped function may modify a location which is updated later on in the symbolic execution. More specifically, a load instruction from address *addr* is *dependent on an invocation of a skipped function* if and only if: (1) *addr* is among the locations that *may* be modified by the skipped function (according to the may-mod analysis), and (2) no stores to that location happened between the skipped invocation function and the load. In particular, when the second condition does not hold, no recovery is needed as the stores performed by the skipped function are irrelevant. mayMod() utilises the information gathered during the symbolic execution regarding overwritten locations (algorithm 1, line 42) to refine on-the-fly the detection of the *relevant* side effects of skipped functions.

3.2 Multiple Recovery States

In some cases, we need to create several recovery states during a single chopped symbolic execution.

For example, consider the following code fragment which replaces lines 7 to 12 of the `main()` function in Figure 2:

```
7  f(&p,k); // skip
8  // next two branches depend on the side effects of f
9  if (p.x)
10   p.z++;
11 if (p.y)
12   p.z--;
```

If we wish to skip the invocation to `f()` then a recovery state and a dependent state are created at each of the branches on lines 9 and 11. Note that the second dependent state is produced from the first dependent one and that the resumed state encapsulates the changes made by the first recovery state. Assume that these changes involve a modification of the value of `p.x` inside the $k > 0$ branch at line 20. If the symbolic execution of the second recovery state goes through the path in which `p.y` is updated ($k \leq 0$), the induced combined execution would be infeasible. To avoid this undesirable situation, when a recovery state terminates, it adds the new constraints accumulated in its path condition to the *guiding constraints* of its dependent state. The added constraints are then used in subsequent recovery states. In our example in Figure 2, the constraint $k > 0$ is propagated from the first recovery state to the first dependent state, thus ensuring that the symbolic execution of the second recovery state does not follow an infeasible path.

```
1  struct point { int x, y;};
2  void f1(struct point *p) {
3      p->y = 1;
4  }
5  void f2(struct point *p) {
6      if (p->y)
7          p->x = 1;
8  }
9  void g() {
10     struct point p;
11     f1(&p); // skip
12     f2(&p); // skip
13     if (p.x) {
14         // ...
15     }
16 }
```

Figure 3: Multiple skipped functions.

3.3 Handling Multiple Skipped Functions

So far, we have assumed that every symbolic state has at most one skipped invocation. When multiple invocations are skipped and more than one may modify the dependent load address *addr*, we need to decide which functions to use for recovery and in which order. We solve this issue by executing the skipped invocations according to their order along the path, thus ensuring that the value stored in *addr* at the end of the recovery process is indeed the last value written there along the chopped path.

Another issue that we need to address to support multiple skipped functions is that a skipped invocation might depend on the side effects of an earlier skipped function. When this happens, we apply our recovery approach in a recursive manner, and treat the current recovery state as a *dependent* state. For example, consider the code in Figure 3. When the execution reaches the dependent load at line 13, we create a recovery state for `f2`, since `f1` does not modify the field x. When the created recovery state reaches the load instruction at line 6, it identifies it as a dependent load. Chopped symbolic execution then creates another recovery state which executes `f1`. Once the recovery of `f1` is terminated, we can continue with the recovery of `f2`.

To make the symbolic execution more efficient in these cases, we maintain for each state a *recovery cache*. The *recovery cache* records for each skipped invocation and slice, the resulting values which were written by the skipped function during the recovery process. This enables us to avoid re-executing the recovery process in certain cases. For example, if g had read `p->y` after the invocation of `f2`, we could have found the value of `p->y` in the cache.

3.4 Memory Allocations

Let us consider the example from Figure 4, where the skipped function `f` allocates memory with `malloc`. After skipping the function call at line 7, the chopped symbolic execution encounters two dependent loads at lines 8 and 9 and thus spawns two consecutive recovery states: one which executes only line 3 (as line 4 is removed by the static slicer), and one which executes lines 3 and 4. If we allowed `malloc` to return two different addresses while executing the recovery states, this could lead to an incorrect execution since the second recovery would write to a different memory address. To prevent this, and maintain consistency across recovery states

```
struct point { int x, y; } *p = NULL;
void f() {
  p = malloc(sizeof(struct point));
  p->x = 0;
}
void g() {
  f(); // skip
  if (p)
    if (p->x) {
      // ...
  }
}
```

Figure 4: Example of skipped function with allocation.

originating from the same function call, we maintain a list of returned addresses for each allocation site in `f`, which are identified by their call stack. This way, subsequent recovery states will use this information while re-executing allocating instructions. Other nondeterministic routines, such as `random`, can be handled in a similar way.

3.5 Chopping-Aware Search Heuristic

Search heuristics are the main approach to reduce path explosion and steer symbolic execution to uncovered paths for a more effective exploration [10–12, 43], and chopped symbolic execution is no exception. However, these heuristics do not take into account the particular nature of the states in chopped symbolic execution, particularly the distinction between normal and recovery states.

We propose a *chopping-aware search heuristic*, which attempts to optimize the exploration of chopped symbolic execution. The search heuristic favors the selection of normal states, which do not require any recovery, thus fostering code exploration. Since always favoring normal states over recovery states may lead to saturation in code exploration, we allow the searcher to select a recovery state at a lower probability. Through experimentation, we determined that choosing to execute a recovery state with a probability of 0.2 produces the best results, and this is the value we use in our evaluation.

Technically, we maintain two worklists, one for normal states and one for recovery states. When the symbolic execution engine selects the next state to explore, it first selects a worklist according to the specified probability, and only then extracts the next state. As a result, it is possible to manage the two worklists according to different heuristics.

3.6 Limitations

The current main limitation of the technique is related to symbolic addresses. Handling a symbolic address is difficult, as it may refer to multiple allocation sites, which in turn may lead to the recovery of several different skipped functions. Moreover, chopped symbolic execution needs a concrete load address to update when stores are performed to that address during recovery.

Chopped symbolic execution currently focuses on skipping functions. However, the approach is more generic: In theory, we could skip any arbitrary code portion that preserves the control-flow of the program. We are currently working on such an extension, particularly on designing an appropriate API for specifying arbitrary code portions to skip.

4 IMPLEMENTATION

We implemented chopped symbolic execution into Chopper, an extension to the KLEE symbolic execution engine [11]. We make Chopper available at https://srg.doc.ic.ac.uk/projects/chopper/.

We forked KLEE from commit `b2f93ff`. A user can run Chopper by specifying the list of functions to skip along with specific call sites via command-line switches.

Chopper combines static analysis—in particular mod-ref analysis and slicing—with symbolic execution. Since KLEE operates on LLVM bitcode, we rely on libraries that statically analyse LLVM bitcode. In particular, we implemented a library for static analysis that exposes APIs to KLEE, so new or better static analyses can be integrated in Chopper with ease.

We compute *mod-ref analysis* by using the pointer analysis provided by SVF [38]. In particular, we rely on a flow-insensitive and context-insensitive pointer analysis based on the Andersen algorithm [4]. We compute static backward slicing using the DG static slicer [20]. We modified the slicer to be able to generate slices of arbitrary functions and not only of the entry point of the program. Note that static slicing is computed on-demand, when a recovery is required. The same slice may be reused for multiple recoveries, so each slice is computed only once.

5 EXPERIMENTAL EVALUATION

Our evaluation aims to provide preliminary evidence that this novel form of symbolic execution can lead to significant scalability gains. More specifically, we evaluate its effectiveness when embodied in the following two scenarios:

(1) **Failure reproduction**, where the research question we explore is: How does chopped symbolic execution perform with respect to standard symbolic execution in generating an input that triggers a failure? In particular, can it reproduce more failures than standard symbolic execution, or can it reproduce the same failures faster?
(2) **Test suite augmentation**, where the research question we explore is: How does chopped symbolic execution perform when steered to generate test cases that improve the structural coverage of code? Can chopped symbolic execution complement the exploration of standard symbolic execution?

Note that our objective is not to claim that chopped symbolic execution is generally a superior technique for a specific task—and thus omits a direct comparison with other state-of-the-art techniques for each scenario—but rather to assess the attainable benefits of chopped symbolic execution when applied to techniques built upon symbolic execution engines.

We compare Chopper with baseline KLEE. We use the same KLEE commit (`b2f93ff`) from which we based Chopper. Both tools are compiled with LLVM 3.4.2 [29] and use STP 2.1.2 as the constraint solver [22]. We conduct our experiments on servers running Ubuntu 14.04, equipped with an 8-core Intel processor at 3.5 GHz and 16GB of RAM.

5.1 Failure Reproduction

In this experiment we use chopped symbolic execution for failure reproduction. In particular, we run a symbolic executor to generate inputs that trigger known security vulnerabilities.

Table 1: Security vulnerabilities and libtasn1 versions considered for reproduction.

Vulnerability	Version	C SLOC
CVE-2012-1569	2.11	24,448
CVE-2014-3467	3.5	22,091
CVE-2015-2806	4.3	28,115
CVE-2015-3622	4.4	28,109

Benchmarks. The subjects of this part of the evaluation are vulnerabilities taken from GNU libtasn1. As briefly discussed in the introduction, GNU libtasn1 is a library for serialising and deserialising data in Abstract Syntax Notation One (ASN.1) format. For example, libtasn1 is used in GnuTLS to define X.509 certificates. We selected the libtasn1 library because its code is complex, with nested and deep function calls, and can be successfully analysed by the KLEE symbolic executor. Table 1 lists the vulnerabilities selected for our experiment, which are memory out-of-bounds accesses. Note that each vulnerability requires the reproduction of a single failure, except for CVE-2014-3467, for which the vulnerability can be exploited in three different code locations, so we consider three different failures. Therefore, in this experiment we aim to reproduce a total of six failures.

Methodology. We proceed with the following evaluation process:

(1) We manually create an execution driver for the libtasn1 library to exercise the library from its public interface, simulating the interactions of an external program (e.g., GnuTLS).
(2) We manually derive the set of functions to skip by inspecting the code and the vulnerability report which usually includes the stack trace and sometimes results from a dynamic analysis tool (e.g., Valgrind [31]). For the selected case studies we managed to identify a candidate set of function to exclude in less than 30 minutes per failure, but a developer familiar with the code should be able to do so faster.
(3) We invoke KLEE and Chopper on the subject with several different search heuristics for normal states (random, DFS, and coverage-based[7]) and DFS for recovery states. We use a timeout of 24 hours. We also configure the symbolic executors to terminate the execution as soon as the vulnerability is identified. We do that by adding a new option to KLEE that, given a list of code locations, terminates the execution as soon as a vulnerability is discovered at all locations.

Results. Table 2 summarises the high-level results of our failure reproduction experiment. For each vulnerability and search heuristic we report the effectiveness of KLEE and Chopper at reproducing the failure as the time required to generate an input that triggers the vulnerability.

As can be seen, Chopper outperforms KLEE on all but one case study both in terms of number of failures reproduced and performance, regardless of the search heuristic applied. Overall, KLEE reproduces four failures, CVE-2014-3467_1, CVE-2014-3467_2, CVE-2015-2806 and CVE-2015-3622, and only failure CVE-2014-3467_1 can be reproduced with all search heuristics. This latter case seems

[7]KLEE search options `dfs`, `random-state` and `nurs:covnew` respectively.

Table 2: Results for the failure reproduction experiment on libtasn1. For each case we report either the time required for exploration if the vulnerability was reproduced successfully (in *mm:ss*), Timeout (if execution is terminated after 24 hours), or OOM (if the execution runs out of memory, with the respective time at which this happened).

Vulnerability	Search	KLEE	Chopper
CVE-2012-1569	Random	*OOM (11:52)*	02:27
	DFS	*OOM (05:08)*	03:29
	Coverage	*OOM (11:28)*	02:45
CVE-2014-3467_1	Random	00:05	00:45
	DFS	16:31	00:08
	Coverage	00:03	00:58
CVE-2014-3467_2	Random	1:02:13	06:18
	DFS	*Timeout*	00:09
	Coverage	1:33:56	02:48
CVE-2014-3467_3	Random	*Timeout*	09:55
	DFS	*Timeout*	12:31
	Coverage	*Timeout*	09:50
CVE-2015-2806	Random	1:07:46	02:18
	DFS	2:46:13	12:04
	Coverage	*OOM (38:56)*	01:02
CVE-2015-3622	Random	*Timeout*	00:16
	DFS	*Timeout*	18:41
	Coverage	20:25:20	00:18

to be relatively easy to identify, since KLEE requires only a few seconds. On the other cases, KLEE requires between 1 and 20 hours. The problem of path explosion in KLEE is particularly visible in CVE-2012-1569 where the symbolic executor quickly runs out of available memory (4096 MB) and thus fails to reproduce the failure.

In contrast, Chopper can identify all vulnerabilities and generates a test case to reproduce each failure in less than 20 minutes, and often much faster. Overall, for the vulnerabilities that KLEE can also reproduce, Chopper can significantly beat KLEE in terms of performance by at least an order of magnitude, with the only exception of CVE-2014-3467_1 where Chopper can be slowed by the cost of static analyses.

Table 3 summarises the detailed results of Chopper for the failure reproduction experiment. For each vulnerability and search heuristic we report the number of snapshots and recovery states generated during chopped symbolic execution (Snapshots and Recoveries, respectively), the execution times for Chopper with and without slicing (Sliced $\mathcal{F}$ and Full $\mathcal{F}$, respectively) as well as statistics on the generated slices, which includes the number of slices generated (Num), and the total size of the original ($\mathcal{F}$ size) and sliced ($\mathcal{S}$ size) skipped functions in terms of LLVM instructions.

Table 3 shows that the number of skipped function calls (as deduced by the number of snapshot states) and recovery states varies with the nature of the case study, the skipped functions, and the search heuristic. In the case of vulnerability CVE-2015-2806, Chopper could reproduce the failure without recovering. This is the exemplar case that highlights the benefits of chopped symbolic execution: While KLEE spent hours interpreting code unrelated

Table 3: Detailed results of Chopper for the failure reproduction experiment on libtasn1.

Vulnerability	Search	Snapshots	Recoveries	Function		Slice		
				Full $\mathcal{F}$	Sliced $\mathcal{F}$	Num	$\mathcal{F}$ Size	$\mathcal{S}$ Size
CVE-2012-1569	Random	5,315	7,447	01:21	02:27	4	694	320 (46%)
	DFS	381	1,078	00:10	03:29	4	694	320 (46%)
	Coverage	6,258	9,053	01:53	02:45	4	694	320 (46%)
CVE-2014-3467$_1$	Random	6,607	6,883	00:30	00:45	4	3,740	2,318 (62%)
	DFS	656	1,003	00:07	00:08	4	3,740	2,318 (62%)
	Coverage	5,642	7,357	00:50	00:58	4	3,740	2,318 (62%)
CVE-2014-3467$_2$	Random	16,279	26,300	07:38	06:18	4	3,740	2,318 (62%)
	DFS	656	1,003	00:11	00:09	4	3,740	2,318 (62%)
	Coverage	10,147	18,916	04:43	02:48	4	3,740	2,318 (62%)
CVE-2014-3467$_3$	Random	26,762	43,480	17:30	09:55	4	3,740	2,318 (62%)
	DFS	38,696	61,113	17:07	12:31	4	3,740	2,318 (62%)
	Coverage	30,947	42,797	17:13	09:50	4	3,740	2,318 (62%)
CVE-2015-2806	Random	173,065	0	02:31	02:18	-	-	-
	DFS	2,708,849	0	12:30	12:04	-	-	-
	Coverage	36,549	0	01:04	01:02	-	-	-
CVE-2015-3622	Random	584	8,980	00:25	00:16	6	1,269	343 (27%)
	DFS	23,846	20,188	21:24	18:41	7	1,453	398 (27%)
	Coverage	608	9,043	00:23	00:18	6	1,269	343 (27%)

with the failure, Chopper excluded the uninteresting code portions and could proceed analysing only code of interest, consistently identifying the failure with all search heuristics in as little as one minute.

Table 3 also shows that the benefit of slicing the skipped functions depends on the case study. For example, for the CVE-2014-3467$_3$ vulnerability, Chopper is on average 70% faster when slicing the skipped functions. Conversely, Chopper performs the best without slicing in CVE-2012-1569. A plausible explanation is that the additional analyses required for slicing were more expensive than directly analysing the functions. We plan to develop a lightweight analysis to speculatively identify when to apply slicing on the skipped functions.

5.2 Test Suite Augmentation

In this experiment we use chopped symbolic execution for test suite augmentation. We do that by running Chopper on a subject program where we skip functions already exercised by an existing test suite. As initial test suite we rely on tests generated by KLEE. In essence, we want to assess the effectiveness of chopped symbolic execution in complementing standard symbolic execution in test generation, for the goal of increasing structural coverage.

Benchmarks. The subjects of this part of the evaluation are GNU BC 2.27, LibYAML 0.1.5, and GNU oSIP 4.0.0. BC[8] is an arbitrary-precision calculator that solves mathematical expressions written in a C-style language. LibYAML[9] is a well-known library for parsing and emitting data in YAML format, which is a human-friendly data serialisation standard. oSIP[10] implements the SIP protocol and provides an interface for creating SIP based applications. We choose these benchmarks because KLEE has a hard time generating high-coverage tests. As a result, the code not covered by KLEE is usually related to complex features, and we challenge Chopper to exercise it. For each program, we rely on the program's documentation and personal experience with the subject to identify the best argument configuration that can maximise coverage.

Methodology. We proceed with the following evaluation process:

(1) We generate the initial test suite by running KLEE on each subject with the coverage-based search heuristic and a time limit of one hour. We use this configuration to maximise structural coverage of the code under analysis, in particular we focus on line and branch coverage.
(2) We compute the structural coverage obtained with the test suites that KLEE generates using GNU GCov.[11]
(3) We use the coverage information and the call graph to select for each program the set of functions to skip. For example, suppose that function f calls function g and h, and that f and h are covered by a test. We include in the set of the skipped functions only h, since f is required to reach uncovered function g.
(4) We invoke Chopper on the subjects with the coverage-based search heuristic for normal states and DFS for the recovery states. We use a timeout of one hour.

Results. Table 4 summarises the results of our test suite augmentation experiment. For each case study we report the structural coverage of a symbolic executor as percentages of lines and branches covered by its generated test suite. For KLEE+Chopper we report the structural coverage results with and without performing slicing (Sliced $\mathcal{F}$ and Full $\mathcal{F}$, respectively).

Table 4 shows that Chopper effectively complements KLEE and increases code coverage even on complex subjects. Specifically, on

[8]https://www.gnu.org/software/bc
[9]https://pyyaml.org/wiki/LibYAML
[10]https://www.gnu.org/software/osip
[11]https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Table 4: Line (L) and branch (B) coverage achieved by KLEE and KLEE+Chopper for BC, LibYaml and oSIP in one hour.

Program	KLEE		KLEE+Chopper Full $\mathcal{F}$		KLEE+Chopper Sliced $\mathcal{F}$	
	L	B	L	B	L	B
BC	23.2%	15.6%	26.6%	19.9%	27.2%	20.8%
LibYAML	10.8%	4.2%	19.9%	11.5%	19.9%	11.5%
oSIP	5.7%	5.1%	9.8%	8.7%	9.8%	8.7%

BC, Chopper increased statement and branch coverage by 4% and 5.2%, respectively; on LibYAML it approximately doubled coverage; and in oSIP it also led to significant gains.

In BC, Chopper managed to skip expensive functions that initialize the parsing of the input file and reached the actual parsing functions. Unfortunately, the analysis quickly got stuck in the parsing routine due to timeouts in the constraint solver, resulting in a limited increase in coverage.

In the case of LibYAML we observed that KLEE spent almost all its budget analysing one function that contains complex logic responsible for ensuring that the buffer contains enough characters for parsing while handling different encodings, such as UTF-8 or UTF-16. This function is invoked at the beginning of program execution, and KLEE got stuck in it, not being able to execute any subsequent line of code. Conversely, Chopper skipped the expensive invocation and continued to explore other parts of the code. Our chopping-aware search heuristic also allowed us to recover paths inside the expensive function while giving higher priority to non-recovery states, in turn resulting in a more in-depth exploration of the code.

A similar scenario was encountered in oSIP, where KLEE spent a considerable amount of resources on a white character processing routine which is invoked at the beginning of the execution. By skipping this routine, Chopper was able to perform a deeper exploration of the code.

As for the previous experiment, the benefit of slicing strictly depends on the case study. In this experiment, slicing is not beneficial in LibYAML and oSIP, while it leads to increased coverage in BC.

5.3 Threats to Validity

Here we briefly discuss the countermeasures we adopted to mitigate the threats to validity. The internal validity depends on the correctness of our prototype implementation, and may be threatened by the evaluation setting and the execution of the experiments. We carefully tested our prototype with respect to the original KLEE baseline, and make it available for further inspection.

Threats to external validity may derive from the selection of benchmarks. We validated our approach on three real-world subjects. Different results could be obtained for different subjects. The only way to further reduce the external validity threat consists in replicating our study on more subjects. For this reason we make our experimental package publicly available to other researchers.[12]

[12]https://srg.doc.ic.ac.uk/projects/chopper/

6 RELATED WORK

The research community has invested significant effort in addressing the path explosion challenge in symbolic execution, and this paper aligns with this line of work.

As we already mentioned in the introduction, the most common and often most effective mechanism employed by symbolic executors are search heuristics, whose goal is to guide program exploration to the most promising paths in the program. Popular heuristics include random path exploration [11], generational search [25] and coverage-optimized search [10, 12], to name just a few. Unfortunately, search heuristics only partly alleviate path explosion, and symbolic execution can still get stuck in irrelevant parts of the code.

Another effective technique is to try to prune equivalent program paths [8, 9]. For instance, if a path reaches a program point with a set of constraints equivalent to those of a previous path that reached that point, then the second path (and all paths that it would have spawned) can be terminated. This technique is similar in spirit to our approach, but orthogonal, as it does nothing to prevent the exploration of code irrelevant to the task at hand. Chopped symbolic execution can be combined with path pruning, in order to prune both irrelevant paths, as well as those relevant paths which are equivalent to other relevant paths.

Merging paths can also help alleviate path explosion. Paths can be merged either ahead-of-time [17, 18] or at runtime [28, 36]. A particular type of path merging are function summaries, in which paths within a function are merged into a summary that can be reused on subsequent invocations [2, 23]. Path merging can lead to exponential reduction in the number of paths explored, but the cost is often offloaded to the constraint solver, which has to deal with significantly harder constraints. Again, chopped symbolic execution could be combined with path merging, in order to get the benefit of both.

Chopped symbolic execution makes use of program slicing in order to explore only the relevant parts of code through the skipped functions. Program slicing has been explored in symbolic execution before, e.g., in the context of patch testing [6].

7 CONCLUSION

Chopped symbolic execution is a novel form of symbolic execution which allows users to specify uninteresting parts of the code that can be excluded during analysis, thus focusing the exploration on those paths most relevant to the task at hand. Our preliminary evaluation shows that chopped symbolic execution can lead to significant improvements in scalability for different scenarios such as vulnerability reproduction and test suite augmentation. Future work can explore these scenarios and others in more depth, aim to further automate the identification of functions to skip, and extend the approach with the ability to skip arbitrary code fragments.

Acknowledgements. We thank the ICSE reviewers, Frank Busse and Timotej Kapus for their useful comments on the paper. This research was supported in part by the UK EPSRC through grants EP/N007166/1 and EP/L002795/1, by Len Blavatnik and the Blavatnik Family foundation, Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University and the Pazy Foundation.

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[2] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, Mar.-Apr. 2008.

[3] S. Anand, C. S. Păsăreanu, and W. Visser. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *Proc. of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'07)*, Mar.-Apr. 2007.

[4] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[5] A. Aquino, F. A. Bianchi, C. Meixian, G. Denaro, and M. Pezzè. Reusing constraint proofs in program analysis. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)*, July 2015.

[6] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'11)*, July 2011.

[7] D. Binkley and M. Harman. A survey of empirical results on program slicing. *Advances in Computers*, 62:105–178, 2004.

[8] P. Boonstoppel, C. Cadar, and D. Engler. RWset: Attacking path explosion in constraint-based test generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[9] S. Bugrara and D. Engler. Redundant state detection for dynamic symbolic execution. In *Proc. of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, June 2013.

[10] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. of the 23rd IEEE International Conference on Automated Software Engineering (ASE'08)*, Sept. 2008.

[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, Dec. 2008.

[12] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: Automatically Generating Inputs of Death. In *Proc. of the 13th ACM Conference on Computer and Communications Security (CCS'06)*, Oct.-Nov. 2006.

[13] C. Cadar, P. Godefroid, S. Khurshid, C. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic Execution for Software Testing in Practice—Preliminary Assessment. In *Proc. of the 33rd International Conference on Software Engineering, Impact Track (ICSE Impact'11)*, May 2011.

[14] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, 2013.

[15] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on binary code. In *Proc. of the IEEE Symposium on Security and Privacy (IEEE S&P'12)*, May 2012.

[16] V. Chipounov and G. Candea. Reverse engineering of binary device drivers with revnic. In *Proc. of the 5th European Conference on Computer Systems (EuroSys'10)*, Apr. 2010.

[17] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of floating-point and SIMD code. In *Proc. of the 6th European Conference on Computer Systems (EuroSys'11)*, Apr. 2011.

[18] P. Collingbourne, C. Cadar, and P. H. Kelly. Symbolic crosschecking of data-parallel floating-point code. *IEEE Transactions on Software Engineering (TSE)*, 40(7):710–737, 2014.

[19] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[20] DG website. https://github.com/mchalupa/dg.

[21] I. Erete and A. Orso. Optimizing constraint solving to better support symbolic execution. In *Proc. of the Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA'11)*, Mar. 2011.

[22] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Proc. of the 19th International Conference on Computer-Aided Verification (CAV'07)*, July 2007.

[23] P. Godefroid. Compositional dynamic test generation. In *Proc. of the 34th ACM Symposium on the Principles of Programming Languages (POPL'07)*, Jan. 2007.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'05)*, June 2005.

[25] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Proc. of the 15th Network and Distributed System Security Symposium (NDSS'08)*, Feb. 2008.

[26] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proc. of the 2nd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, June 2001.

[27] X. Jia, C. Ghezzi, and S. Ying. Enhancing reuse of constraint solutions to improve symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'15)*, July 2015.

[28] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'12)*, June 2012.

[29] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*, Mar. 2004.

[30] T. Liu, M. Araújo, M. d'Amorim, and M. Taghdiri. A comparative study of incremental constraint solving approaches in symbolic execution. In *Proc. of the Haifa Verification Conference (HVC'14)*, Nov. 2014.

[31] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[32] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program repair via semantic analysis. In *Proc. of the 35th International Conference on Software Engineering (ICSE'13)*, May 2013.

[33] H. Palikareva and C. Cadar. Multi-solver support in symbolic execution. In *Proc. of the 25th International Conference on Computer-Aided Verification (CAV'13)*, July 2013.

[34] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar. Accelerating array constraints in symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'17)*, July 2017.

[35] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, Sept. 2005.

[36] K. Sen, G. Necula, L. Gong, and W. Choi. Multise: Multi-path symbolic execution using value summaries. In *Proc. of the joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'15)*, Aug.-Sept. 2015.

[37] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Found. Trends Program. Lang.*, 2(1):1–69, Apr. 2015.

[38] Y. Sui and J. Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *Proc. of the 25th International Conference on Compiler Construction (CC'16)*, Mar. 2016.

[39] N. Tillmann and J. De Halleux. Pex: white box test generation for .NET. In *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, Apr. 2008.

[40] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[41] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proc. of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'12)*, Nov. 2012.

[42] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[43] T. Xie, N. Tillmann, J. Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *Proc. of the 2009 International Conference on Dependable Systems and Networks (DSN'09)*, June 2009.

[44] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2):1–36, Mar. 2005.

[45] G. Yang, C. S. Păsăreanu, and S. Khurshid. Memoized symbolic execution. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'12)*, July 2012.