Lumberjack: Cutting the Tree
An introduction to three state space explosion mitigations in symbolic execution

Seminar Paper

ambiso

***Abstract*—Symbolic execution is a powerful technique to generate test inputs of arbitrarily complex functions, check if programs violate their model's properties or assertions, find inputs that lead to desired states or aid in the process of automatic exploit generation. The method is haunted by the inevitable predicament of the state space explosion: attempting to discover all feasible paths in a program in a sound and complete way is undecidable in general and must entail acute caveats such as non-termination or absurd memory requirements. We present three techniques that attempt to mitigate the damage and ameliorate the applicability of the scheme to complex software-components by minimizing the number of states that, to retain soundness, must be explored.**

Keywords*—*Symbolic Execution, Redundant State Detection, S²E, Path Partitioning, State Space Explosion

I. Introduction

Symbolic execution is a powerful method for the analysis of programs aiding in test case generation[1], bounded model checking[2] and automatic exploit generation[3]. The technique was introduced in the mid'70s for debugging, testing and to falsify program assertions[4]–[6]. In recent years a plethora of symbolic execution engines have sprung into life[1], [3], [7]–[10]. Black-box approaches to automated testing quickly reach their limits, as demonstrated in [1] using a simple example:

```
int foo(int x) { // x is an input
  int y = x + 3;
  if (y == 13) abort(); // error
  return 0;
}
```

If the underlying machine uses 32 bit integers, the probability of hitting the error branch with a uniformly distributed input `x` is tiny: 2^{-32}.

Symbolic execution offers an alternative: by using the implementation and lifting it into abstract symbolic states we can symbolically execute the function, forming a tree structure of all possible executions. When we've reached the relevant branch we can use satisfiability modulo theories (SMT) solvers to check whether the branch condition $y = 13$ may hold true for any x, w.r.t. the conditions accumulated along the path.

Unfortunately, symbolic execution rapidly becomes infeasible, since unbounded loops may produce infinitely many states, yielding what's referred to as a state space explosion. We investigate *sound* mitigations to the state space explosion problem, where sound means that the symbolic execution remains sound: if a path is found by the analysis it's in fact reachable. Additionally, a *complete* analysis would find all feasible paths through a program for a given start state.

II. Background

Similar to [11], we define symbolic states as the triple $(\text{instr}, \sigma, \pi)$ each defined as follows:

- instr: The next instruction to execute. For simplicity, we restrict instructions to assignments, conditional branches and jumps.
- σ: The symbolic memory, mapping addresses or program variables to symbolic or concrete values. A symbolic value λ is defined in terms of arbitrary first order logic formulas constraining its value.
- π: The path constraints collected along the path to the currently executed instruction. To explore all states, we may set $\pi = \top$ at the beginning of the analysis.

Depending on the current instruction symbolic execution performs different actions: For assignments $x := e$ we evaluate the expression e in the current state and obtain e_s with which we update the symbolic memory σ. Encountering a branch `if b then` $\texttt{p}_1$ `else` $\texttt{p}_2$, we split the symbolic state into two states: a state $C_\top$, in which we assert the branch condition evaluated in the current state b_s to hold and where we execute p_1 next: $C_\top = (p_1, \sigma, \pi \wedge b_s)$, and the dual state $C_\perp$ where we assert $\neg b_s$ to hold, and execute p_2 next: $C_\perp = (p_2, \sigma, \pi \wedge \neg b_s)$.

Symbolic execution incurs 4 major issues consolidated in [11].

- Constraint Solving: Through SMT solvers symbolic execution engines can concretize an input, i.e. find an input satisfying the path constraints a state is subject to. SMT solving is undecidable in general, depending on the underlying theories used.
- Environment: Ways to handle the interaction with the environment, i.e. parts of the system outside of the analyzed unit. For example, when the unit performs a system call to write to a file, the symbolic execution engine needs to manage the

interaction. A simple approach is to concretize the arguments and dispatch the system call to the system, however, this yields inconsistencies and becomes unsound as paths on different branches can interact.

Memory
: Symbolic execution engines may handle pointers, arrays and similar complex data-structures in different ways. For example, we could, when writing to a symbolic address, over-approximate the memory and clobber the entire symbolic memory. Future reads from the memory yield unconstrained values, affecting the soundness of the analysis.

State Space Explosion
: The problem of determining all possible paths through a program is undecidable in general. Symbolic execution strives to be a sound and complete analysis technique, at the cost of potential non-termination. Loops can cause an exponential increase of the number of states in the size of the input space. Analyses may reduce the number of states by eliminating redundant states w.r.t. an equality metric relevant for the task at hand[12].

These issues and their solutions are major contributors to a successful symbolic execution engine. We will focus on the fourth issue: the state space explosion.

A. Concolic execution

When using pure symbolic execution, at each branch, its necessary to check whether the collected path constraints are satisfiable, by dispatching them to the SMT solver. *Concolic* execution, the term a portmanteau of concrete and symbolic, mixes the concrete and symbolic execution. A common approach to concolic execution, termed Dynamic Symbolic Execution, is to use a concrete execution to drive the symbolic execution. Starting with random concrete inputs, the concrete execution will efficiently decide satisfiability for us.

III. Input Partitioning

A concolic approach from 2009 is described in [13]. They partition the symbolic input of the program by exploiting the independence of different parts of the program input. As opposed to the traditional security definition of non-interference [13], define two inputs to be *non-interferent* when there are no data or control dependencies between them.

If an instruction i reads a write to location w_1 and writes to location w_2, w_2 is data dependent on w_1. A write w_2 is control dependent on another write, if a branch that reads w_1 dynamically controls whether w_2 is performed. These dependencies are transitive and span the transitive closure of the described direct dependencies. Examples of the dependencies are illustrated in figure 1.

The non-interferent inputs are identified by partitioning the input. A partition of a set S is a set of disjoint sets (*blocks*) whose union is again the set S. The input partitions can be used to generate inputs independent of the other partitions, minimizing the number of test-cases that must be generated to achieve coverage of every branch.

```
  y = 1 // w1
i{x = y // w2
```

(a) Data Dependency

```
x = 1 // w1
if (x) {
  y = 1 // w2
}
```

(b) Control Dependency

Fig. 1: Data and control dependencies can be tracked by reasoning about the executed path.

Their algorithm *FlowTest* is run on a program and an initial optimistic partition of the set of input variables, where each variable is its own block. This optimistic partition would enable the highest degree of independence, and thus the highest reduction in number of explored paths and generated tests.

The algorithm then performs test generation and iteratively merges the blocks of the partition. The test generation entails concolic execution and concretization of symbolic variables. Additionally it's responsible for keeping track of the data and control dependencies, and maintaining a *flow map*, which stores for each variable the set of input blocks in the current partition that may influence it.

The flow map is obtained through a technique known as dynamic slicing, that identifies the instructions that may mutate a given location. If an entry of the flow map contains multiple input blocks, information may flow between these blocks. We then cannot treat them separately anymore and merge them.

The entire process of test case generation is repeated, the flow map updated, and blocks are merged until convergence.

Majumdar and Xu test their implementation on four binaries, achieving an average coverage of 44%. Their benchmark system was a 2.33 GHz Intel Core 2 Duo with 2 GiB of RAM. Analyzing and averaging their reported metrics they cut down the number of paths by a factor of 3.41 and achieve a speedup of 2.81X.

We will see ways to improve this technique in the following two sections.

IV. Selective Symbolic Execution

S^2E is a concolic execution platform for the implementation of binary analysis tools. It improves upon environment interaction by safely crossing the concrete/symbolic border in both directions[7]. They view the analyzed unit in its environment as part of a system. The environment contains parts of the system not part of the unit. The system is the sum of the unit and the environment.

Each concrete execution is performed in isolation in its own virtual machine. We demonstrate S^2E using the example presented in [7], illustrated in figure 2.

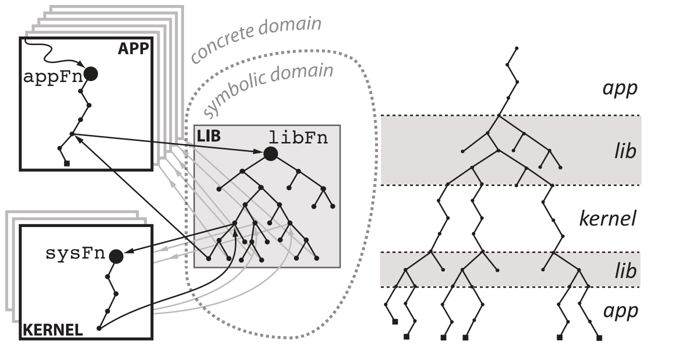


Fig. 2: The app and kernel are part of the environment and concretely executed. The app calls into the lib, which is symbolically executed. Lib calls into the kernel, which causes concrete executions. Shaded regions are symbolically executed. The S^2E execution results in the execution tree on the right. Graphic from [7].

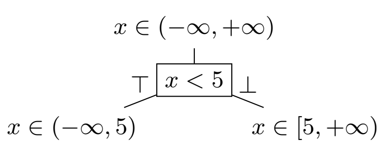
A. From Concrete to Symbolic and Back

When the concretely executed app calls into the symbolically executed lib, e.g. $libFn(10)$ the app receives the return effects of the concretely executed lib-function. Additionally, we explore the lib symbolically. The simplest conversion S^2E offers, is to explore $libFn(\lambda)$ with more general symbolic arguments, instead of the concrete arguments app used.

B. From Symbolic to Concrete and Back

```
void libFn(int x) {
  if (x<5) {
    buf = sysFn(x);
    if (x<0)
    /* ... */
  }
}
```

(a) Example lib function libFn; Example from [7].



(b) Lib function libFn called with a selector converting x to a fully symbolic value. Adapted from [7].

Fig. 3: Lib function example

When calling a function, of which we don't have a model, we need to treat it as a black-box and call it with concretetized arguments. S^2E emulates the concrete execution in a virtual machine and concretizes the arguments lazily: only when the concrete execution has a control dependency on the symbolic value, its concretized. This optimization allows for data to pass through the environment untouched, retaining its symbolic form. S^2E even claim that data may be written to a virtual drive and read back again as symbolically constrained values, without the software stack ever branching on the contents.

Concretizing arguments induces problems when the analysis continues the symbolic execution: if in the libFn, illustrated in figure 3a, `x` was constrained to 4, which is consistent with the path constraints seen in figure 3b, we won't be able to cover the `x < 0` branch. This problem is partially solved by a major contribution by [7] for sound state space reduction: soft constraints. Arguments that were concretized during this type of concrete execution are marked as soft constrained to the values they were assigned. When the execution returns to the symbolic execution and a branch that was possible prior to the concrete call is now blocked, we can opt to go back to a node in the tree of the symbolic execution, where the blocking values were given their values, fork another isolated subtree and choose values satisfying the branch that we want to cover. However, since the concrete execution is a black-box we cannot guarantee that this strategy will succeed. In fact there is a simple counterexample, illustrated in figure 4, that on some systems is impossible to succeed at.

```
1 int libFn(int x) {
2   char *buf = malloc(x);
3   if (buf && x<0) {
4     /* ... */
5   }
6 }
```

Fig. 4: A branch that may be difficult to cover.

In the demonstrated function the memory allocation function `malloc`, which takes as argument the size of the requested memory region and returns a pointer to the first element of the allocated region, is called. We then check if the allocation was successful, by determining if $buf \neq 0$. We may choose to concretize `x` to 1, and reach the branch condition $\texttt{buf} \neq 0 \wedge \texttt{x} < 0$ in line 3, but don't cover line 4, since $\texttt{x} \geq 0$. In fact, under the assumption the emulated system's `malloc`, returns `NULL` for requests of extremely large memory regions, we will not be able to cover the branch, indifferent to the value we concretize `x` to. The assumption is reasonable, unless the system's memory allocator is over-provisioning: Since the `int` is converted to the machine size type: `size_t`. The smallest `size_t` that is also a negative `int` on a 64 bit machine is 18446744071562067968, which is equivalent to approximately 16 exbibytes.

Additionally, S^2E doesn't offer an advantage for control dependencies on the return values of functions in the environment. Say we abstract away an external library offering the `crc32` function, as seen in figure 5. Although the branch would be possible to execute, S^2E may try many different values in vain to cover the branch. S^2E's approach would reduce to a method analogous to fuzzing.

```
int libFn(char *s) {
  if (crc32(s) == 3638176789) {
    /* ... */
  }
}
```

Fig. 5: S^2E cannot efficiently (without resorting to exhaustive search) find an input to cover the `then` branch if `crc32` is abstracted away.

C. Consistency Models

Through relaxed consistency models S^2E mimic the purpose of unit testing. When there's no requirement for a feasible path to exist to a target state, we can relax the consistency requirements for crossing the symbolic to concrete and back barrier. The approach is sound when we are testing a library, whose precise usage behavior shouldn't exclusively be determined by the environment that the driving application prescribes. Assertion violations or crashes discovered on infeasible paths may be of interest too, as the library should be robust w.r.t. a different control-flow. S^2E offers incremental consistency relaxations with their respective use-cases, such that the exploration's results remain meaningful.

V. Redundant State Detection

Bugrara and Engler propose a method for identifying and pruning states that won't exhibit previously unseen behavior. Their approach intertwines an array of complex analysis techniques and is sound, although they provide no formal proof in the paper[12].

The basic idea of [12] is to identify and eliminate the states that won't cover uncovered instructions. The simplest method to that end checks if a state's constraint set at the k^{th} instruction is equal to a previously recorded snapshot, where a snapshot is the constraint set of a previously explored state. However, this naïve and inefficient approach is too restrictive.

A weaker, yet sufficient, condition is to check if the constraint set of a state at the k^{th} instruction is implied by a snapshot of the same instruction. Intuitively this means the snapshot already covered the instruction with at least as general constraints compared to the state that's currently explored. Additionally, since we are only interested in maximizing coverage, we can restrict ourselves to the constraints over memory locations that affect coverage. [12] determines if a location is relevant for coverage through a static control dependence graph and a dynamic dependence graph which are used to perform dynamic slicing. The static control dependence graph yields information on which branches remain relevant to cover. The dynamic dependence graph contains a multitude of dependencies between memory locations.

A. Relevant Static Branches

A branch is statically relevant if its outcome determines whether an uncovered instruction is reachable. We may identify relevant branches statically through a static control dependence graph. Nodes of this graph are static instructions and edges connect branches with instructions that are controlled by the branch's outcome. A static branch is relevant if there is a path in the static control dependence graph from it to an uncovered instruction. In the example in figure 6 the uncovered line 11 is control dependent on line 10, and the uncovered line 3 is control dependent on line 1 and 2. Therefore the relevant static branches are on line 1, 2 and 10.

```
 1 if (reference_file)
 2   if (stat (...))
 3     error(...); //uncovered
 4   ...;
 5 } else {
 6   if (parse_user_spec(...))
 7     error(...);
 8   ...;
 9 }
10 if (chopt.recurse & preserve_root)
11  ...; // uncovered
```

Fig. 6: Linux utility chown; example from [12] with added static control dependencies for uncovered lines.

B. Dynamic Dependence Graph

The dynamic dependence graph is updated throughout the symbolic execution to contain byte-level writes as nodes and data, control and potential dependencies as edges. By reasoning about the currently executed path, we can determine data and control dependencies. Additionally, if a write w_2 is executed control dependent on w_1, but the branch controlling w_2 is not along the executed path, w_2 is potentially dependent on w_1. Potential dependencies can be identified by reasoning about the executed path and static locations on non-executed paths, which requires a sound interprocedural aliasing analysis. Further optimizations and adjustments are necessary to make the method efficient and sound. Mainly, state matching is implemented efficiently and additional edges must be inserted into the static control dependence graph to retain soundness for when the program contains multiple termination points.

C. Dynamic Slicing

Using the relevant static branches and dynamic dependence graph we can slice the program. Slicing the program yields the set of locations that may affect the coverage of uncovered statements. If a snapshot's constraints w.r.t. the relevant locations are a subset of the state's, we eliminate the state.

Constructing the relevant location set is where lies the power and complexity of redundant state detection. It's uncertain whether the approach is feasible to implement for binaries, as techniques like dependency tracking and slicing isn't easy to perform on binaries. Their reference implementation is based on the KLEE symbolic execution engine, which bases its analysis on LLVM. Decompiling and lifting binaries into LLVM bitcode isn't trivial[14].

The authors of the paper report an average coverage increase of 3.8%. They evaluated their implementation on 66 software-components from the GNU coreutils and achieved an increased speedup greater than 1X for 82% of them. 23 of the 89 possible utilities were removed from their analysis either because of issues with the 64-bit implementation, or because the projects were too small and full coverage was obtained instantly. They report a speedup of 50.5X on average, and 10X in the median.

VI. Related Work

The survey by Baldoni, Coppa, D'Elia, *et al.* in [11] provides an extensive overview over the subject and discusses a large set of approaches used to improve symbolic execution. Researchers, proposing a system similar to the one demonstrated in [12] achieve similar performance metrics with a speedup ranging from 1.02X to 49.56X[15]. In contrast to [13]'s input partitioning, [16] partition the output, and similar to [12] also use data, control and potential dependencies, to cut away paths. Additionally, [16] give rigorous definitions and proofs of their algorithms.

VII. Conclusion

The methods demonstrated report immense improvements in the number of states explored when compared to naive implementations. Unfortunately, only S^2E appears to be available for public use. Wang, Liu, Guan, *et al.* claim to have published their implementation, however we weren't able to obtain a copy[15]. We've contacted the authors of [12] via email, asking if they have published their implementation but received no reply within five weeks. Standardized benchmarks and interfaces or more aggressive open-sourcing may aid the research of symbolic execution.

Four of the mentioned studies depend in part on the same techniques and appear to have similar underlying ideas[12], [13], [15], [16]. It may be possible to combine the ideas, as is common in symbolic execution[11], to attain additional gains in performance.

References

[1] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, 20:20–20:27, Jan. 2012, ISSN: 1542-7730. DOI: 10.1145/2090147.2094081. [Online]. Available: http://doi.acm.org/10.1145/2090147.2094081.

[2] M. Eckert, A. Bianchi, R. Wang, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Heaphopper: Bringing bounded model checking to heap implementation security," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD: USENIX Association, 2018, pp. 99–116, ISBN: 978-1-931971-46-1. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/eckert.

[3] T. N. Brooks, "Survey of automated vulnerability detection and exploit generation techniques in cyber reasoning systems," *CoRR*, vol. abs/1702.06162, 2017. arXiv: 1702.06162. [Online]. Available: http://arxiv.org/abs/1702.06162.

[4] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select – a formal system for testing and debugging programs by symbolic execution," in *Proc. of Int. Conf. on Reliable Software*, Los Angeles, California: ACM, 1975, pp. 234–245. DOI: 10.1145/800027.808445. [Online]. Available: http://doi.acm.org/10.1145/800027.808445.

[5] W. E. Howden, "Symbolic testing and the dissect symbolic evaluation system," *IEEE Transactions on Software Engineering (TSE)*, vol. 3, no. 4, pp. 266–278, 1977, ISSN: 0098-5589. DOI: 10.1109/TSE.1977.231144. [Online]. Available: http://dx.doi.org/10.1109/TSE.1977.231144.

[6] J. C. King, "A new approach to program testing," in *Proc. Int. Conf. on Reliable Software*, Los Angeles, California: ACM, 1975, pp. 228–233. DOI: 10.1145/800027.808444. [Online]. Available: http://doi.acm.org/10.1145/800027.808444.

[7] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," *Acm Sigplan Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[8] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, San Diego, California: USENIX Association, 2008, pp. 209–224. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855756.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: Automatically generating inputs of death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 10:1–10:38, Dec. 2008, ISSN: 1094-9224. DOI: 10.1145/1455518.1455522. [Online]. Available: http://doi.acm.org/10.1145/1455518.1455522.

[10] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*, vol. 00, 2016, pp. 138–157. DOI: 10.1109/SP.2016.17. [Online]. Available: doi.ieeecomputersociety.org/10.1109/SP.2016.17.

[11] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[12] S. Bugrara and D. Engler, "Redundant state detection for dynamic symbolic execution," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13, San Jose, CA: USENIX Association, 2013, pp. 199–212. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535486.

[13] R. Majumdar and R.-G. Xu, "Reducing test inputs using information partitions," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 555–569, ISBN: 978-3-642-02658-4.

[14] A. Dinaburg and A. Ruef, "McSema: Static translation of x86 instructions to llvm," in *ReCon 2014 Conference, Montreal, Canada*, 2014.

[15] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Transactions on Software Engineering*, vol. 43, no. 3, pp. 252–271, 2017.

[16] D. Qi, H. D. T. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, 32:1–32:41, Oct. 2013, ISSN: 1049-331X. DOI: 10.1145/2522920.2522925. [Online]. Available: http://doi.acm.org/10.1145/2522920.2522925.