Imperial College London

KLEE ’21

Automating function selection for patch testing via chopped symbolic execution

J. Ruiz, M. Nowack, A. Zaki, C. Cadar

Work in progress

Motivation

Patch testing:

- programs change all the time
- construct automatic test cases that cover the patch

Symbolic execution is expensive

- path explosion
- constraint solving

Chopper

Chopper: chopped symbolic execution

- based on KLEE
- skip parts of the code
  → functions irrelevant to the patch
- main challenge:
  manual function selection

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,

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

D. Trabish, A. Mattavelli, N. Rinetzky, C. Cadar
Chopped Symbolic Execution, In ICSE ’18

Chopper

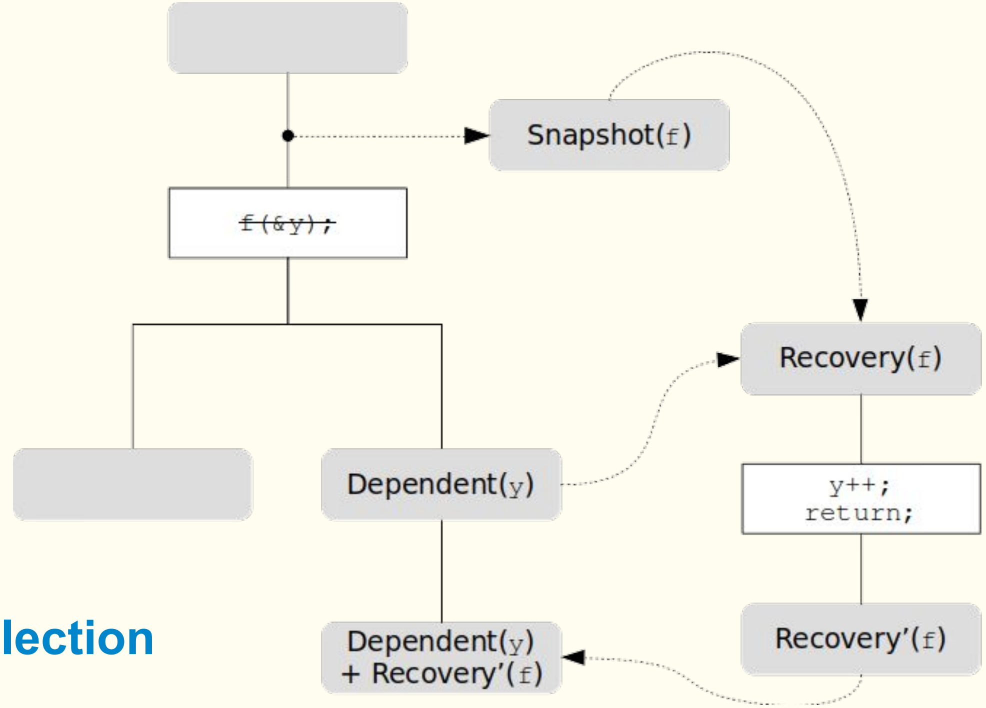
Chopper: lazy execution

- skip functions
- keep snapshots
- monitor dependencies
- trigger recoveries

but…

… recoveries are costly

→ needs **good function selection**



AutoChopper

AutoChopper: Chopper + automatic function selection

Good functions to skip:

+ average sized
+ many forks
+ unlikely to cause recoveries
  + unrelated to patch
  + few side effects (`cleanup()` ✓)
+ no external side effects (`write()` ✗)

AutoChopper

AutoChopper: Chopper + automatic function selection

However…

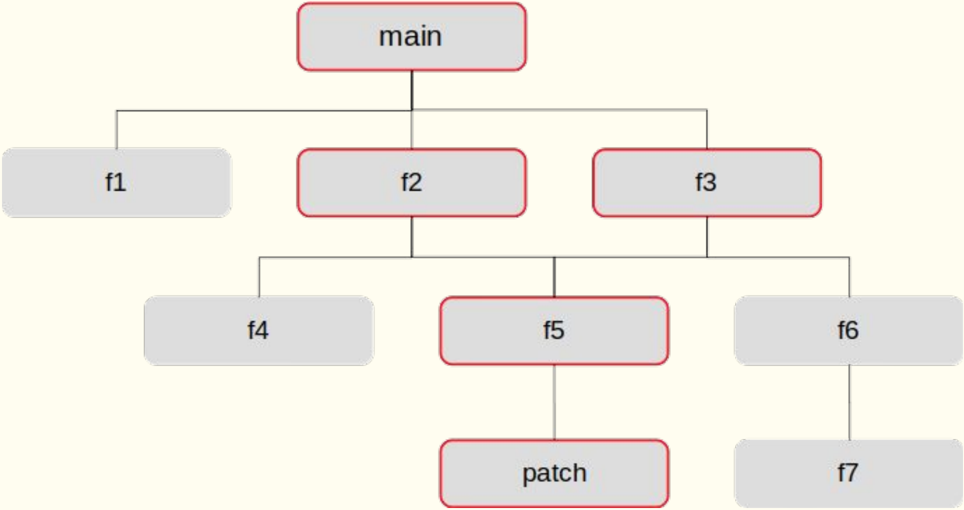
- predicting recoveries is **hard**
- recoveries can be (very) **costly**

- **interrupt** long recoveries
  → **restart** analysis without skipping this function

Automatic function selection

AutoChopper components

- static pass
  lists functions to keep

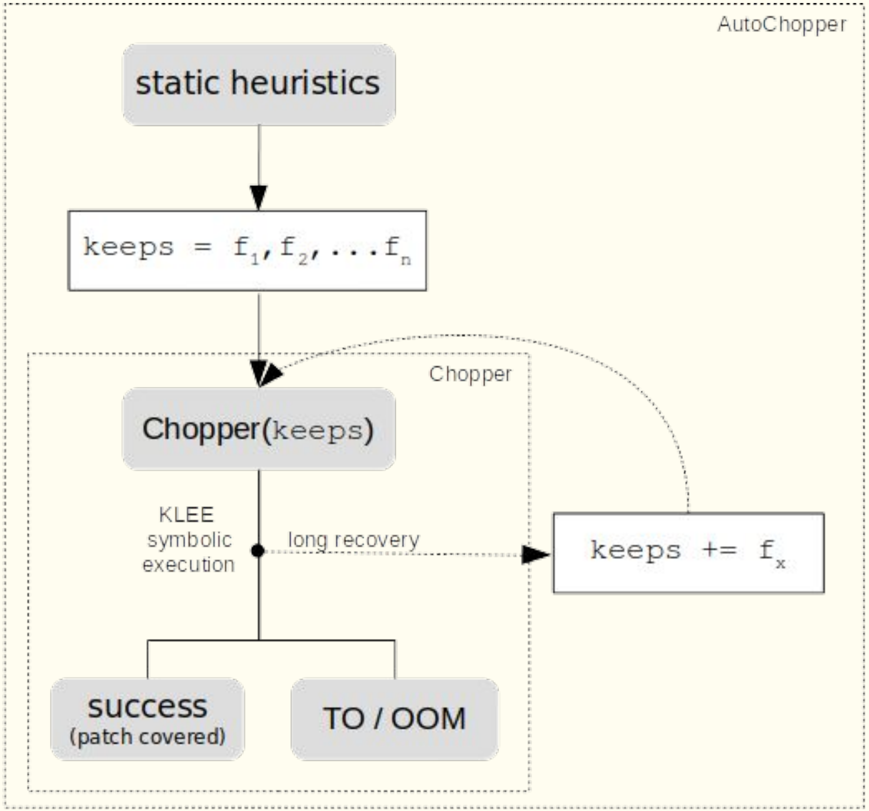


Imperial College London

AutoChopper components

- static pass
  lists functions to keep

- dynamic pass
  restarts analysis

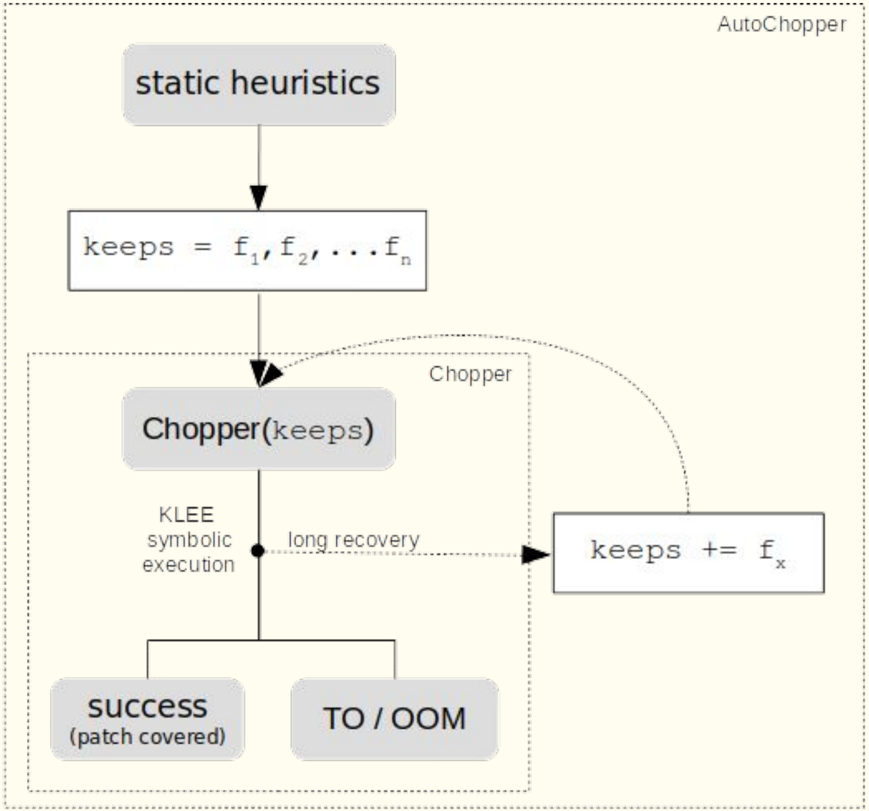


AutoChopper components

- static pass
  lists functions to keep

- dynamic pass
  restarts analysis

- target searcher
  searches for shortest path to the patch



Imperial College London

Experiments: automating Chopper

libtasn1 CVEs - target

Vulnerability	CVE-2012-1569	CVE-2014-3467$_1$	CVE-2014-3467$_2$	CVE-2014-3467$_3$	CVE-2015-2806	CVE-2015-3622
KLEE	05:49	T/O	00:02	T/O	00:57	27:25
Chopper	01:20	06:21	05:41	07:29	01:55	00:21
AutoChopper	02:33	00:47	08:17	01:10	01:12	05:49

AutoChopper performances compare to that of Chopper

Imperial College London

Experiments: automating Chopper

DFS

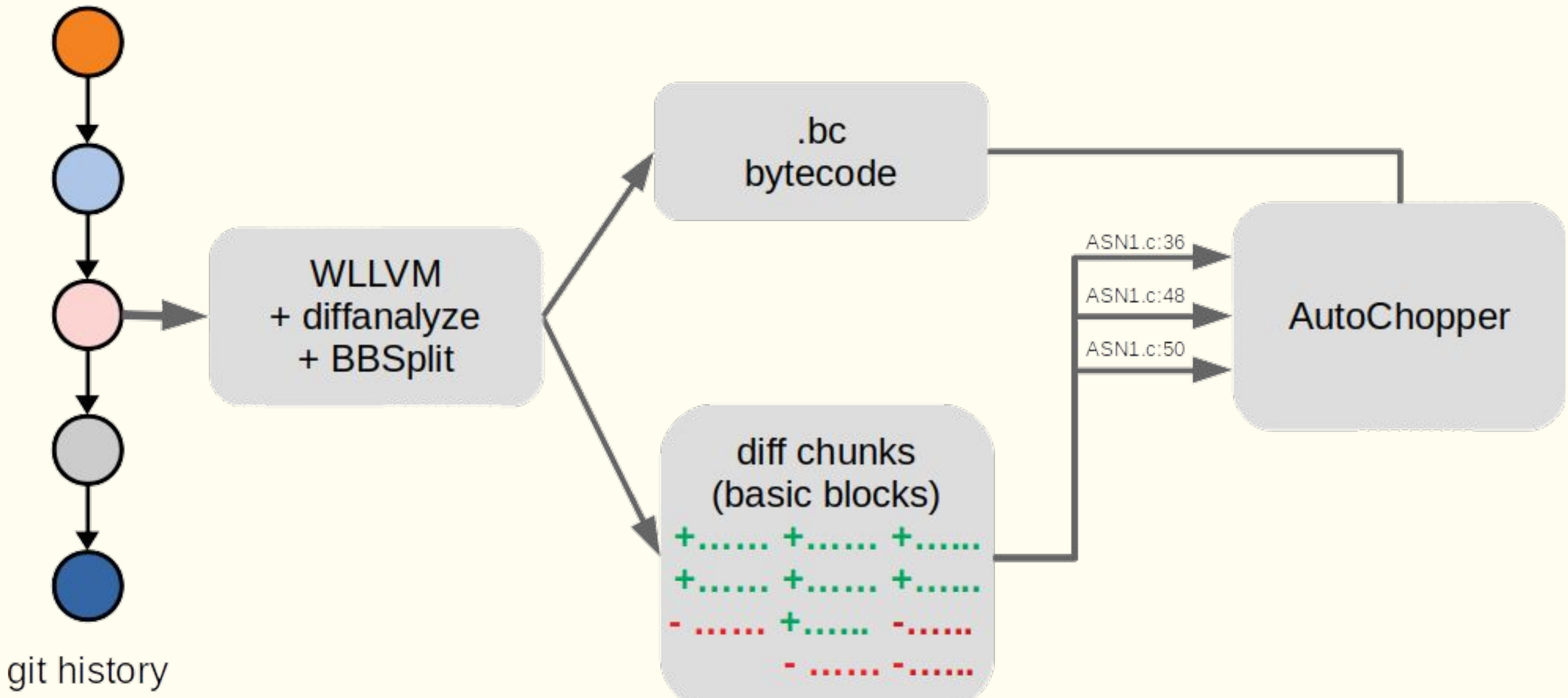
Vulnerability	CVE-2012-1569	CVE-2014-3467$_1$	CVE-2014-3467$_2$	CVE-2014-3467$_3$	CVE-2015-2806	CVE-2015-3622
Chopper	00:31	00:03	01:19	00:06	T/O	12:26
AutoChopper	00:36	00:48	T/O	T/O	00:50	02:04

Coverage

Vulnerability	CVE-2012-1569	CVE-2014-3467$_1$	CVE-2014-3467$_2$	CVE-2014-3467$_3$	CVE-2015-2806	CVE-2015-3622
Chopper	01:39	02:05	05:21	T/O	01:41	00:17
AutoChopper	00:36	00:48	01:14	01:15	00:52	02:59

Patch testing

Imperial College London



Experiments: patch testing

Preliminary results (WiP)

Benchmark	libosip		bc	
	KLEE	AutoChopper	KLEE	AutoChopper
Basic Blocks (commits)	1850 (38)		1691 (374)	
Reachable	982		1636	
Success (over reachables) T/O = 15min	130 13.2%	270 27.5%	146 8.9%	124 7.6%

Conclusion

Automating chopped symbolic execution:

- nearing Chopper’s performances
- some pre-results in automated patch testing

Ongoing challenges

- more benchmarks (libtasn1, libyaml, binutils...)
- preserve debug information better
- rewind symbolic execution instead of restarting

Appendices

Experiments: automating Chopper

libtasn1 CVEs

Vulnerability	CVE-2012-1569				CVE-2014-3467-1				CVE-2014-3467-2				CVE-2014-3467-3				CVE-2015-2806				CVE-2015-3622			
Searcher	Rand	DFS	Cov	Tgt	Rand	DFS	Cov	Tgt	Rand	DFS	Cov	Tgt	Rand	DFS	Cov	Tgt	Rand	DFS	Cov	Tgt	Rand	DFS	Cov	Tgt
KLEE	05:58	01:04	08:43	05:49	T/O	01:13	T/O	T/O	00:03	T/O	**00:02**	**00:02**	T/O	T/O	T/O	T/O	01:06	T/O	**00:44**	00:57	T/O	T/O	27:30	27:25
Chopper	01:05	**00:31**	01:39	01:20	06:12	**00:03**	02:05	06:21	06:31	01:19	05:21	05:41	09:00	**00:06**	T/O	07:29	04:10	T/O	01:41	01:55	00:24	12:26	**00:17**	00:21
AutoChopper	00:35	00:36	00:36	02:33	00:48	00:48	00:48	00:47	01:15	T/O	01:14	08:17	01:19	T/O	01:15	01:10	00:51	00:50	00:52	01:12	02:58	02:04	02:59	05:49

Experiments: patch testing

Preliminary results (WiP)

Benchmark	libosip		libyaml		bc		libtasn1	
	KLEE	ACSE	KLEE	ACSE	KLEE	ACSE	KLEE	ACSE
Basic Blocks (commits)	1850 (38)		2000 (52)		1691 (374)		500 (38)	
Reachable	982		327		1636		162	
Success (over reachables) T/O = 15min	130 13.2%	270 27.5%	6 1.8%	6 1.8%	146 8.9%	124 7.6%	0	0