

UNC CHARLOTTE
College of Computing and Informatics
Department of Software and Information Systems

# Guide Me to Exploit: Assisted ROP Exploit Generation for ActionScript Virtual Machine

Fadi Yilmaz, Meera Sridhar, and Wontae Choi

36th Annual Computer Security Applications Conference (ACSAC'20)
December 7-11, 2020, Online

*presented by*
Fadi Yilmaz

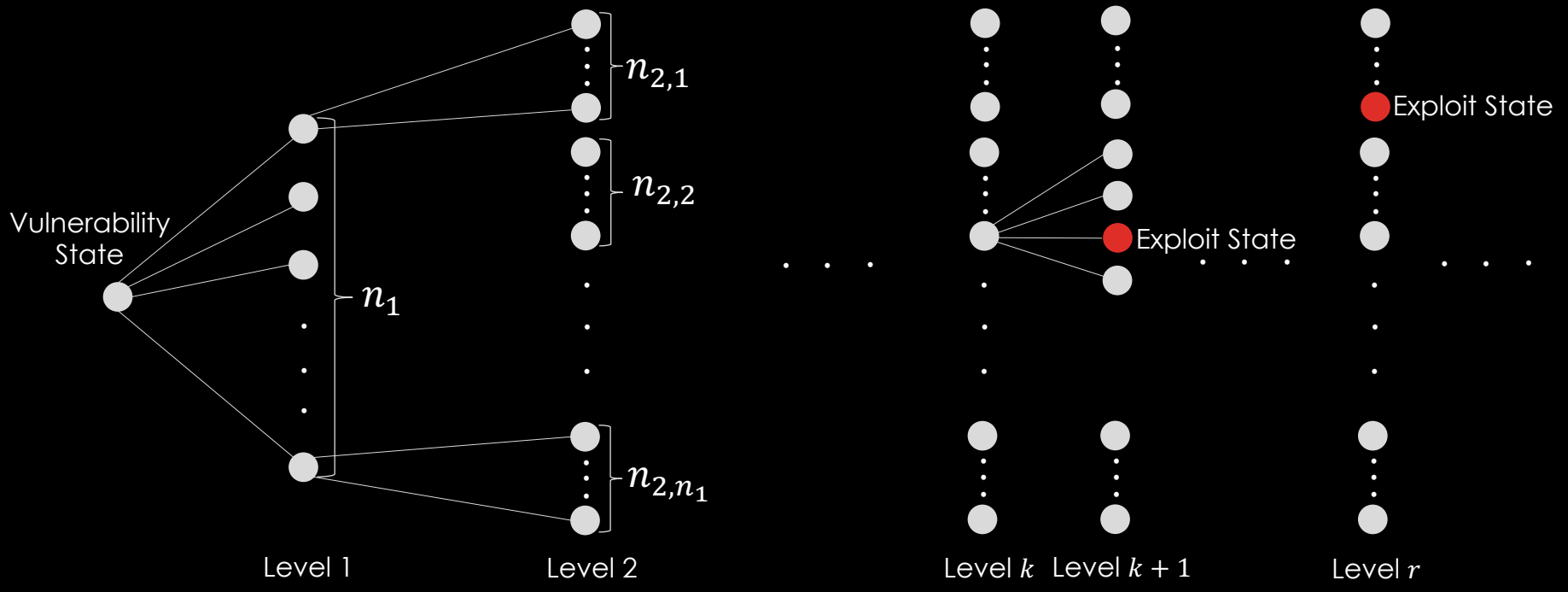
Work supported in part by National Science Foundation Grant No. 1566321

# Motivation of Automated Exploit Generation (AEG)

- Monitoring the execution of exploit scripts is crucial
  - Underlying weaknesses of target applications
  - Unorthodox methods to exploit vulnerabilities

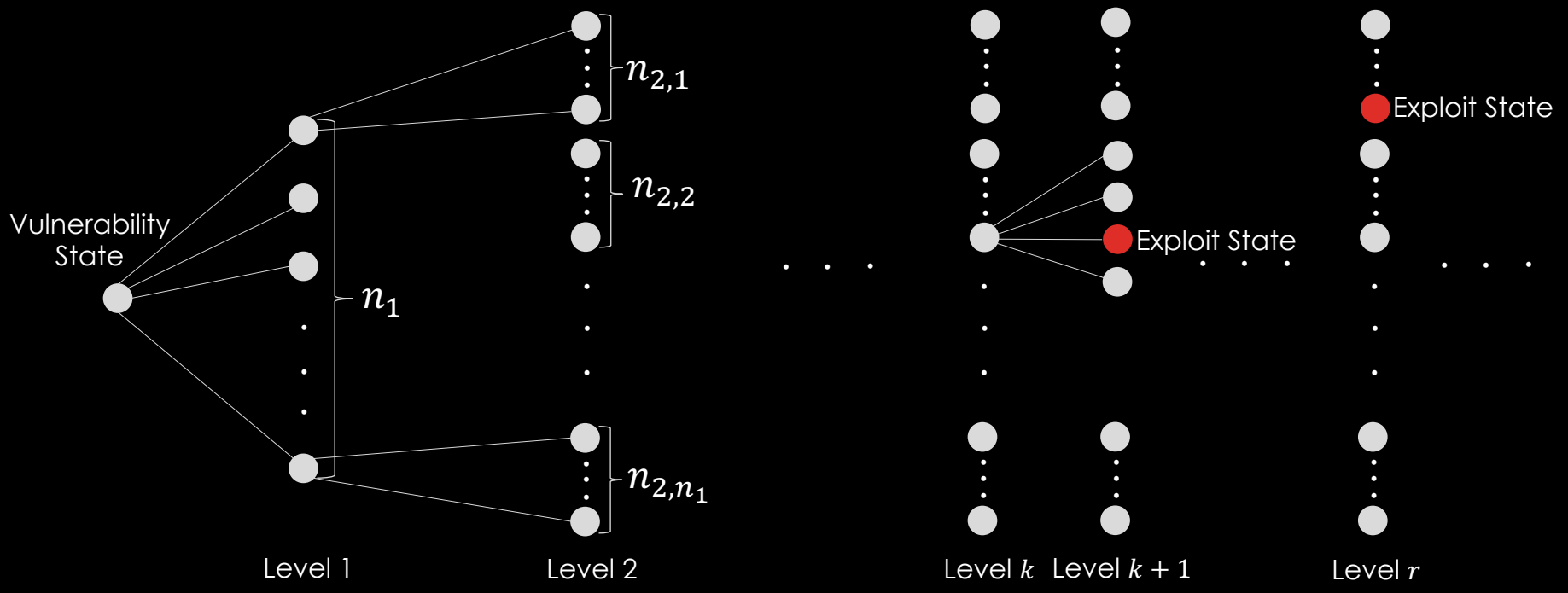
# AEG

- Determining the *exploitability* [Younis et al. SQJ'16]
- Explores all possible execution paths [Avgerinos et al. NDSS'11]



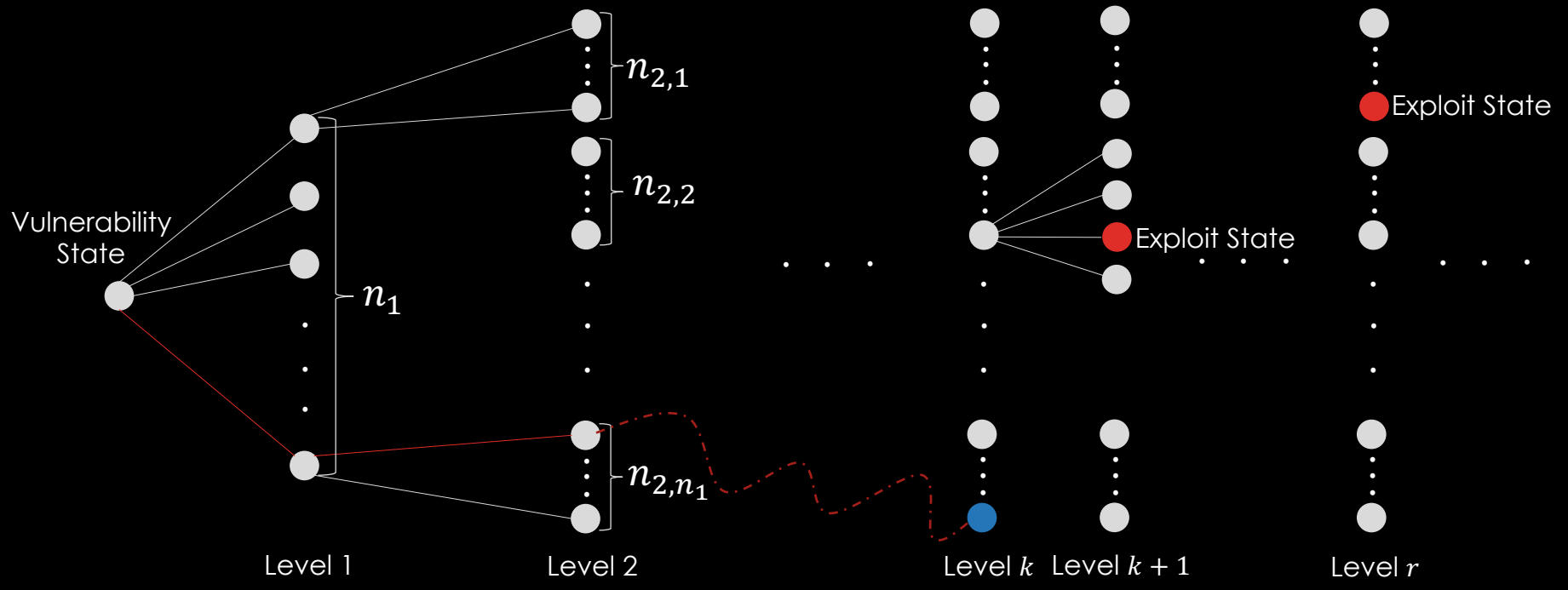
# AEG Components

- Fuzzer [Miller et al. ACM'90, Jayaraman et al. NFM'09, Rawat et al. NDSS'17]
  - Explores only one execution path in one run



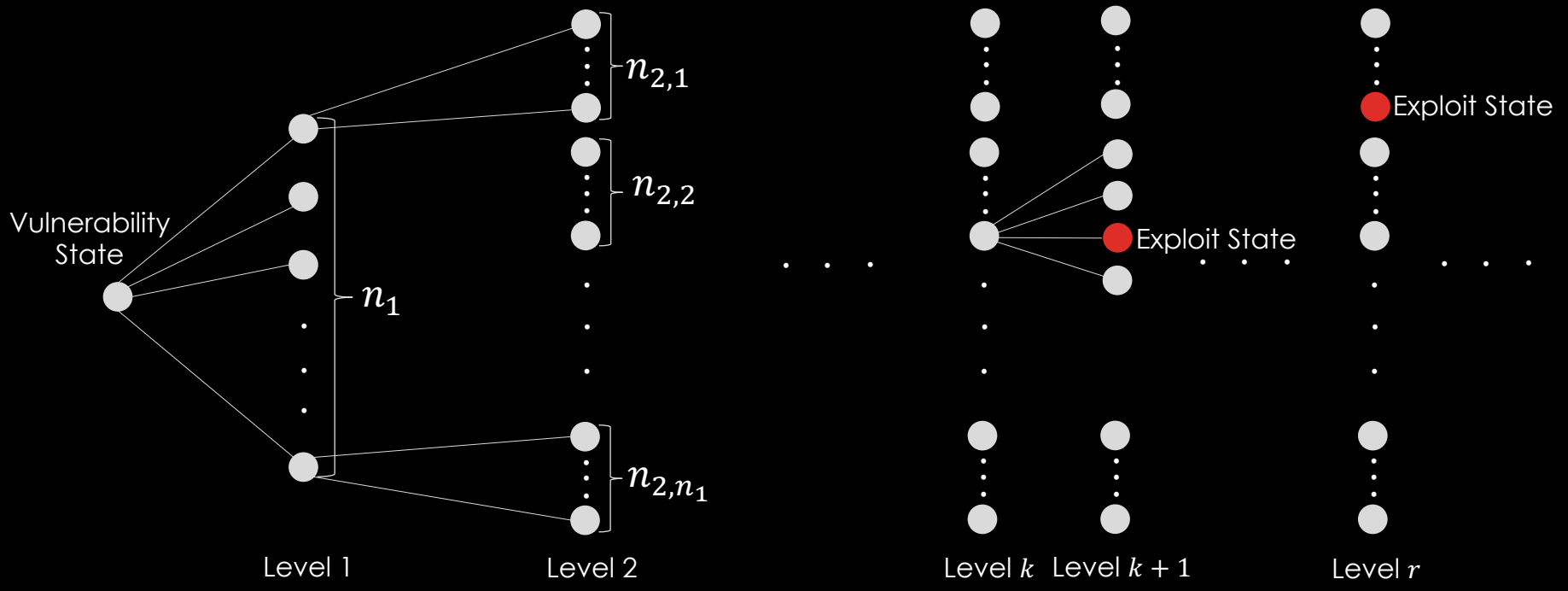
# AEG Components

- Fuzzer
  - Explores only one execution path in one run



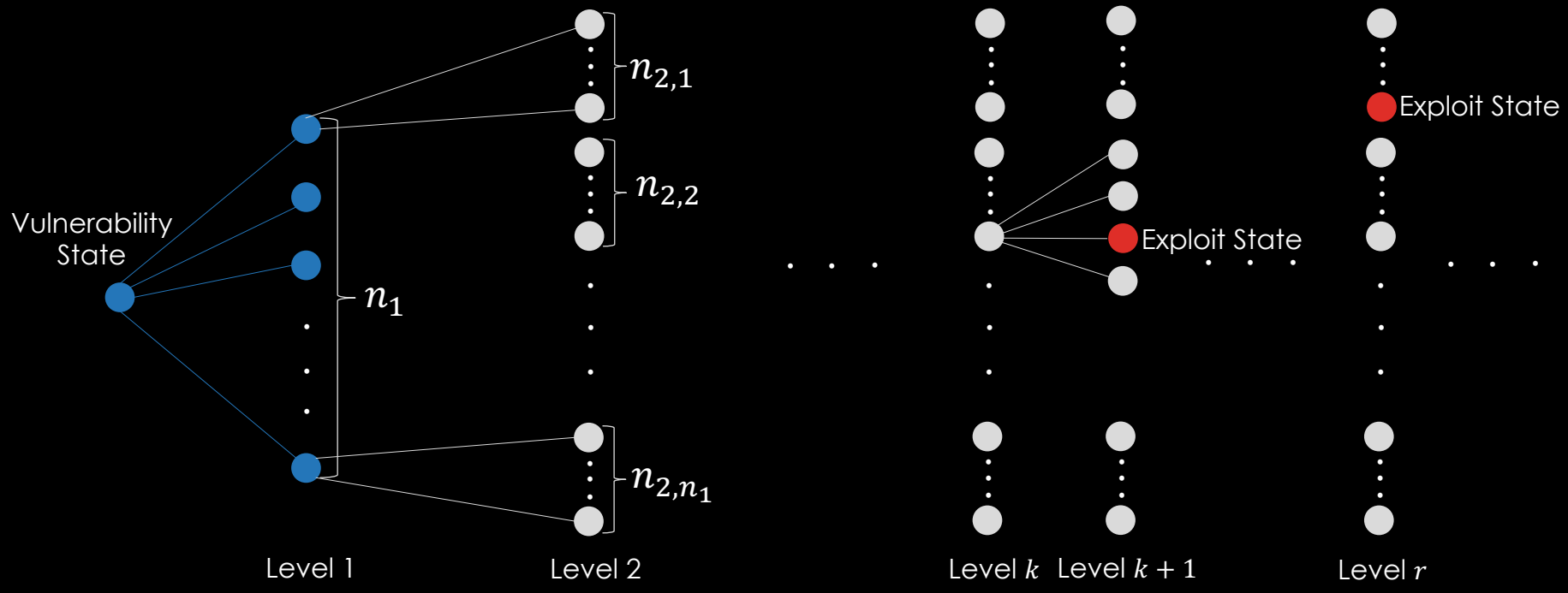
# AEG Components

- Symbolic Execution [King et al. ACM'76]
  - Explores all execution paths symbolically in one run



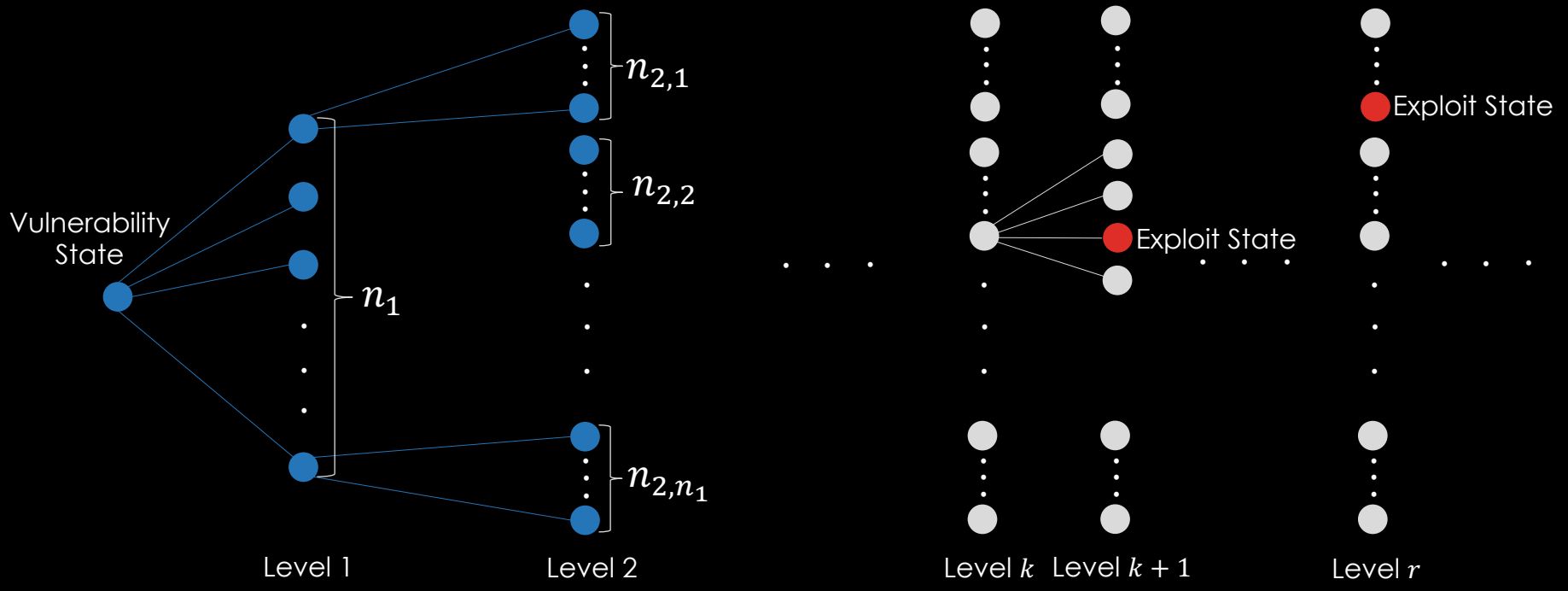
# AEG Components

- Symbolic Execution
  - Explores all execution paths symbolically in one run



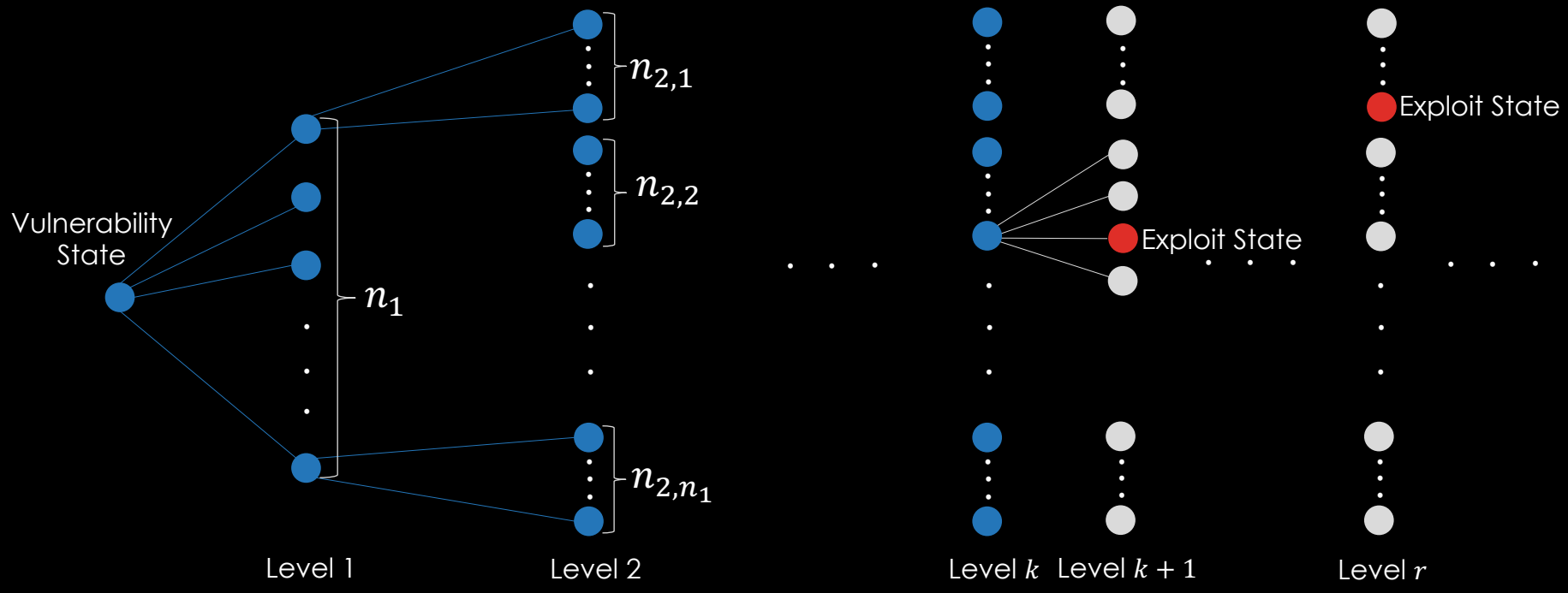
# AEG Components

- Symbolic Execution
  - Explores all execution paths symbolically in one run



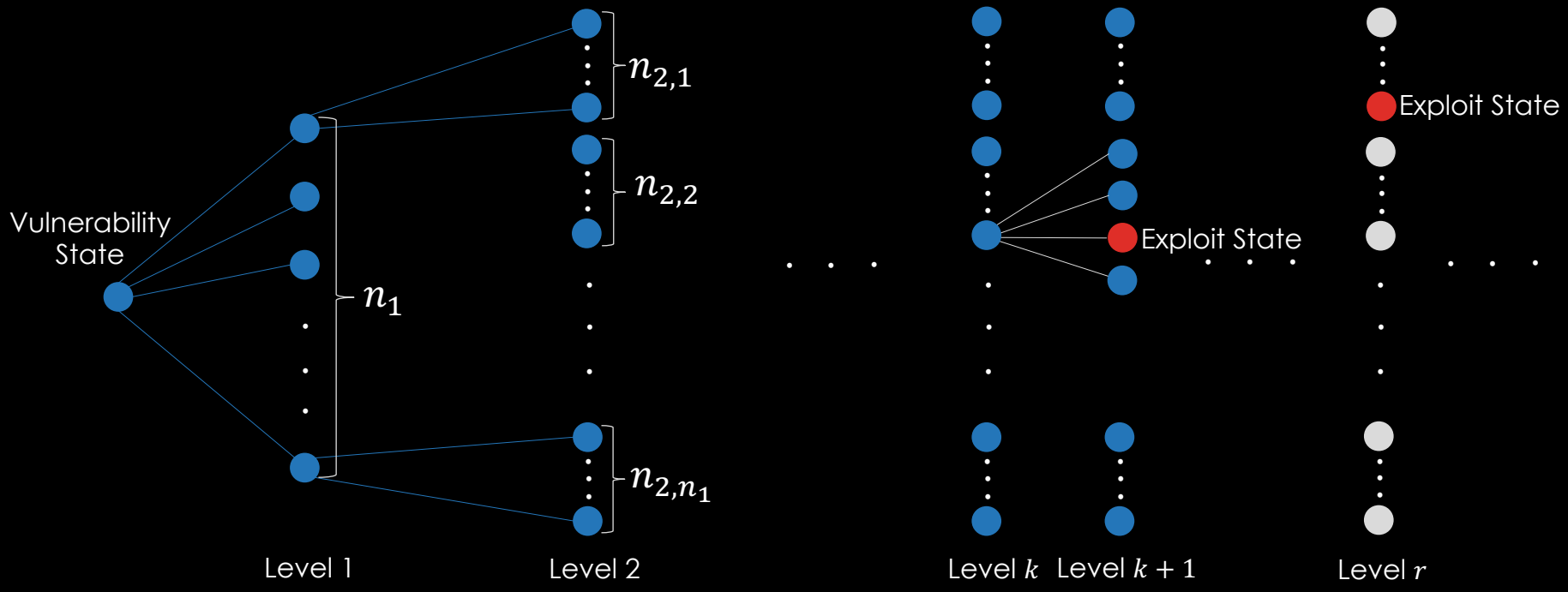
# AEG Components

- Symbolic Execution
  - Explores all execution paths symbolically in one run



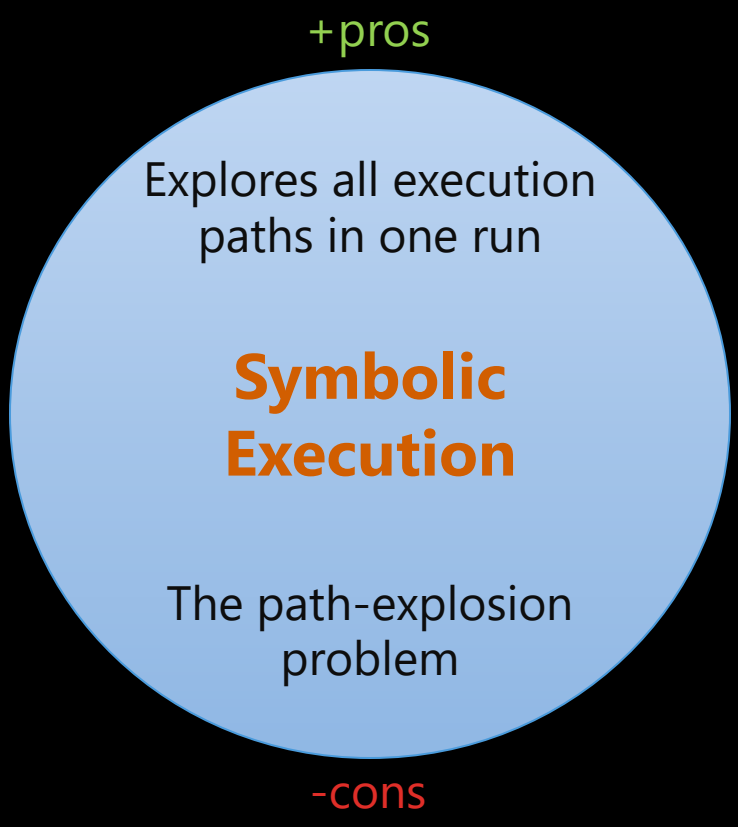
# AEG Components

- Symbolic Execution
  - Explores all execution paths symbolically in one run



# AEG Components



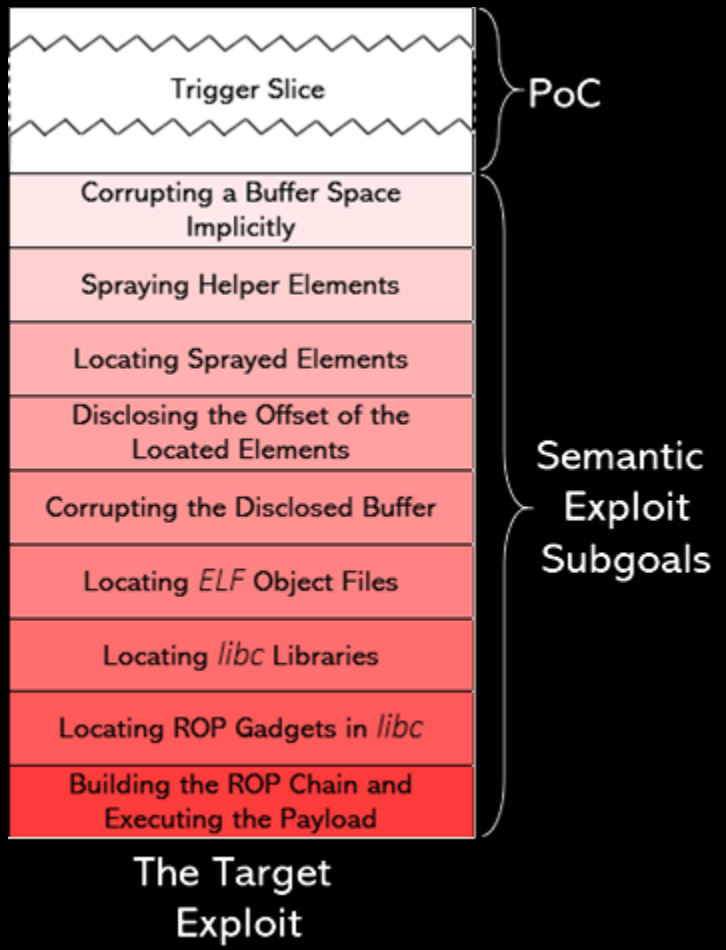



# GUIDEXP : A Prototype Semi-Automatic AEG Tool

- The first *guided* (semi-automatic) exploit generation tool for the AVM implementations
- Does not rely on a fuzzer or a symbolic execution tool

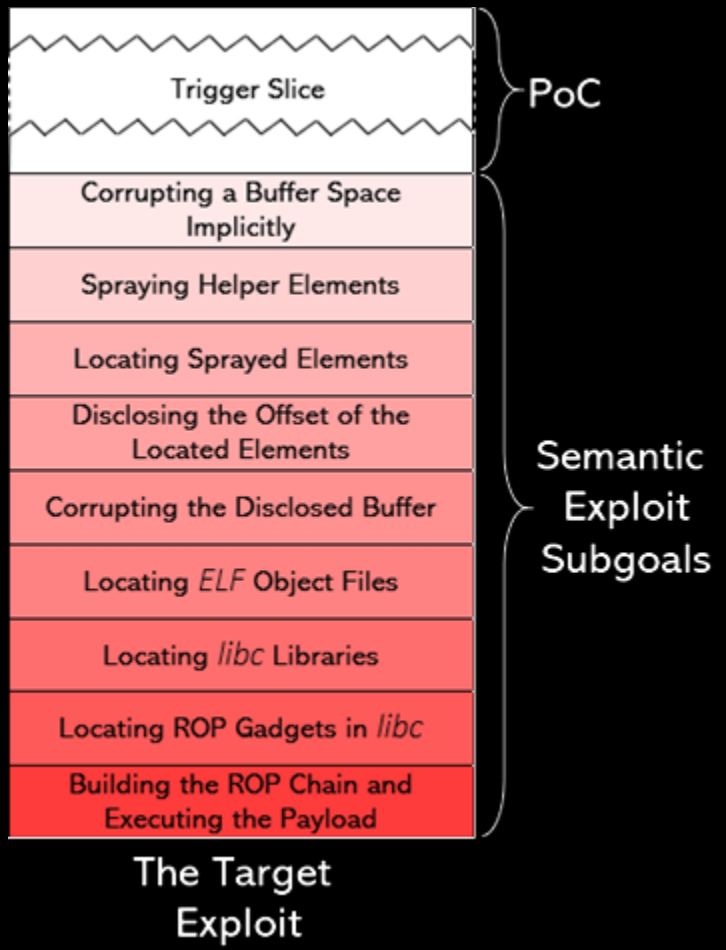
# Intuition Behind Target Exploit Generation

- Structure of our target exploit
- Exploit pattern

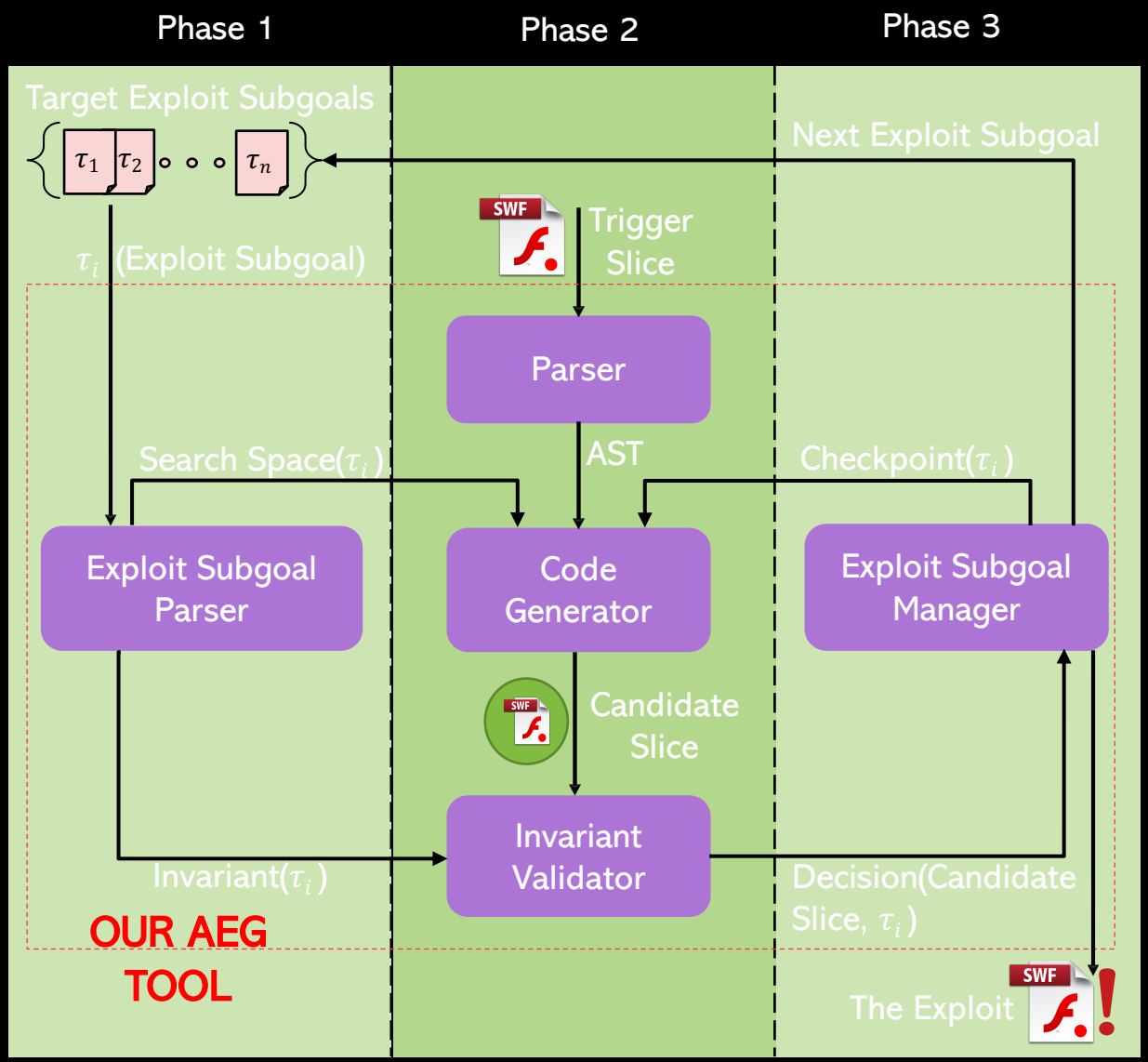


# Exploit Subgoals

- A search space
  - Set of instructions
- An invariant
  - The test

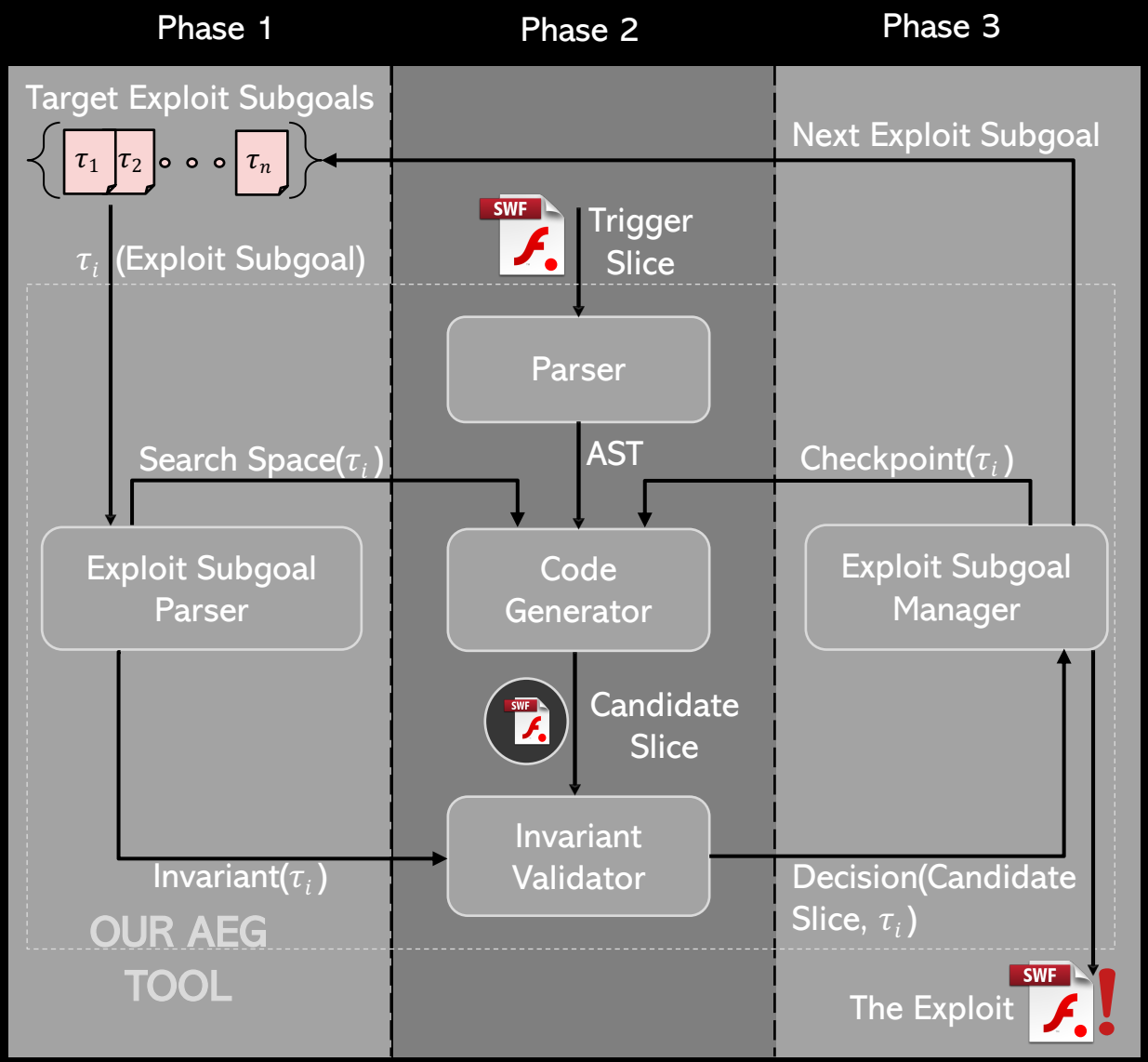


# Preparation: Defining Exploit Subgoals, Inputs & Outputs



# Preparation: Defining Exploit Subgoals, Inputs & Outputs

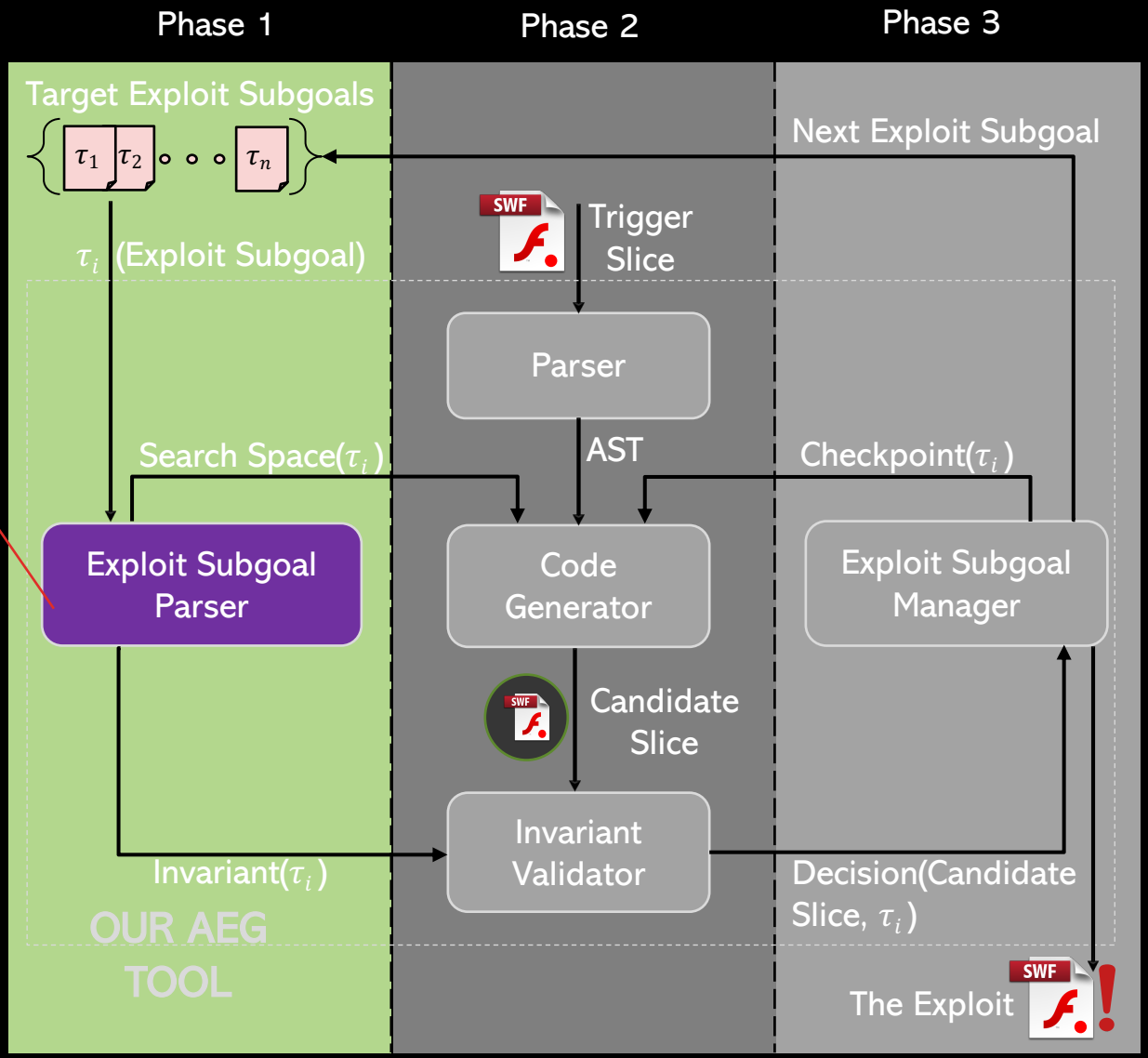




# Phase 1: Exploit Subgoal Processing

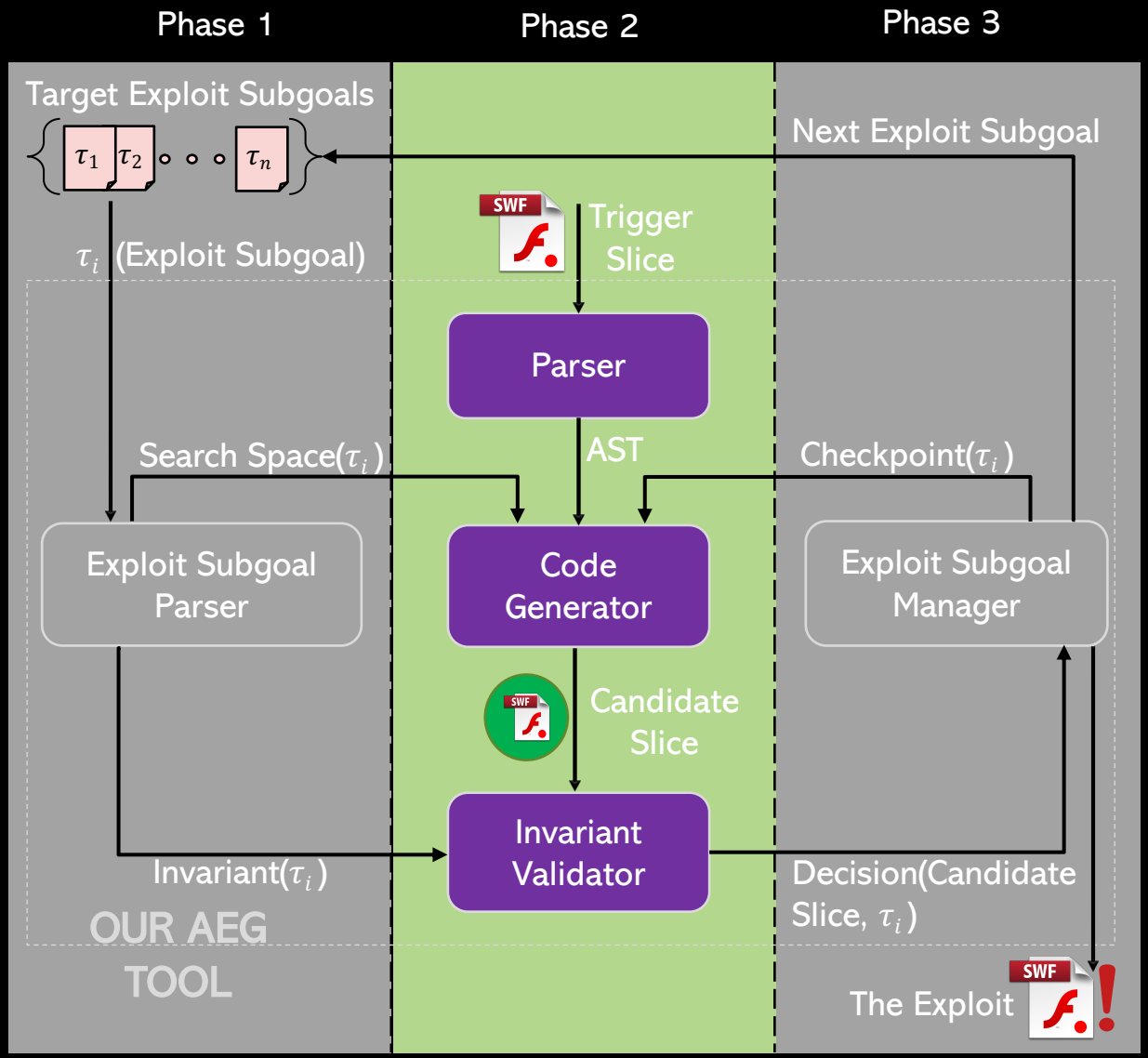


Reads exploit subgoals and invariant



# Phase 2: Generates Candidate Slices and Validating Invariant

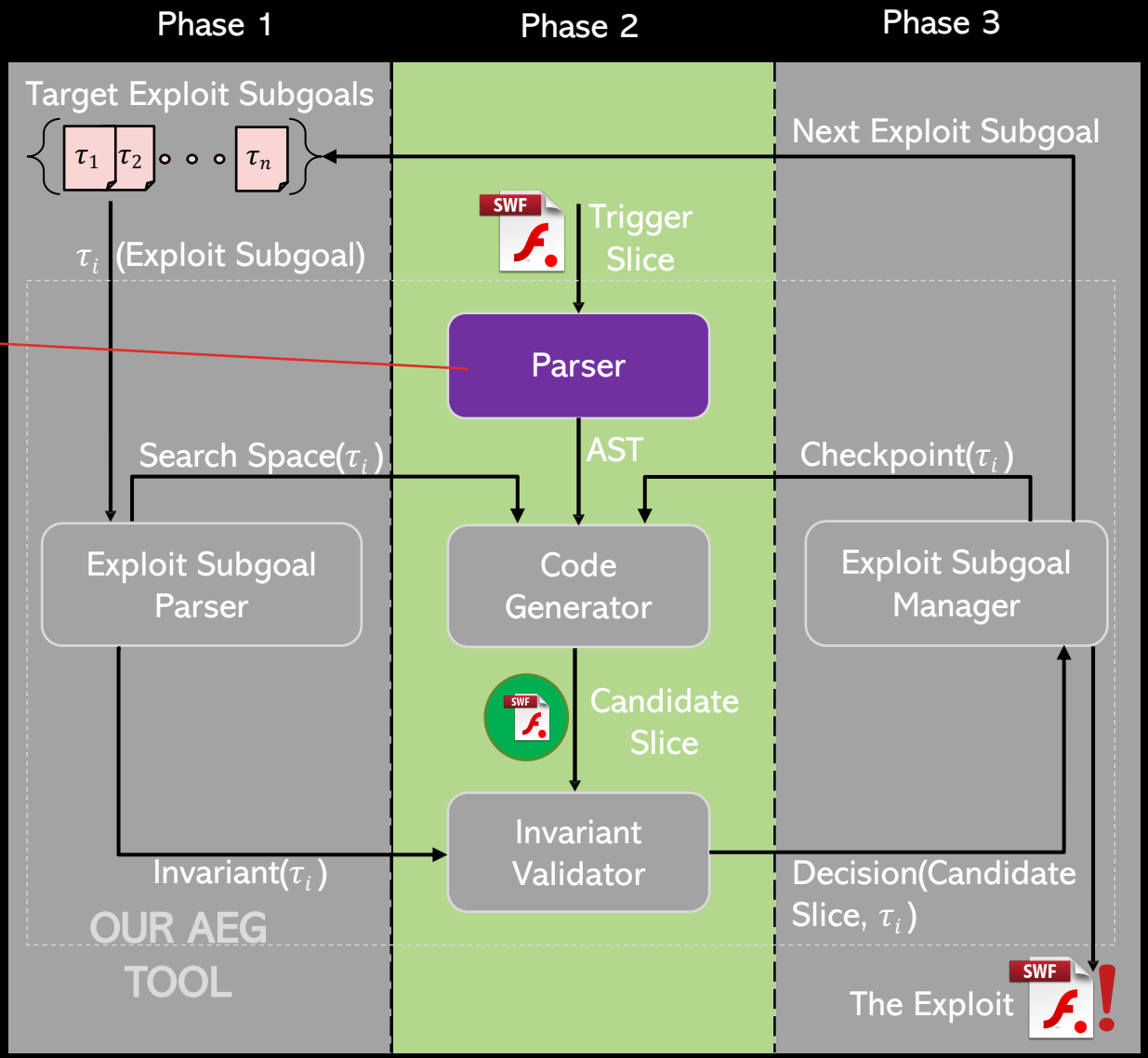




# Phase 2: Generates Candidate Slices and Validating Invariant



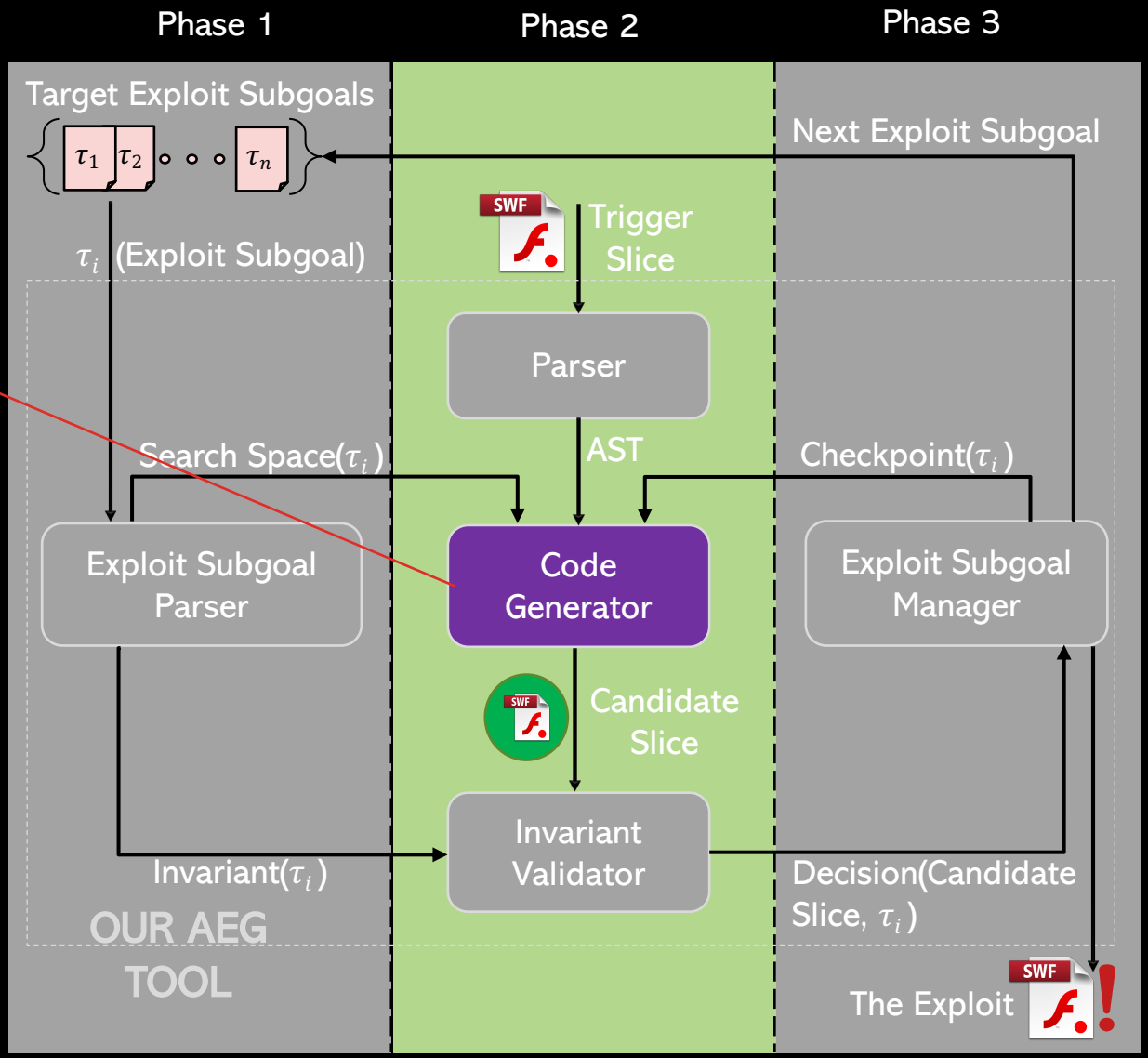
Parses the trigger slice



# Phase 2: Generates Candidate Slices and Validating Invariant



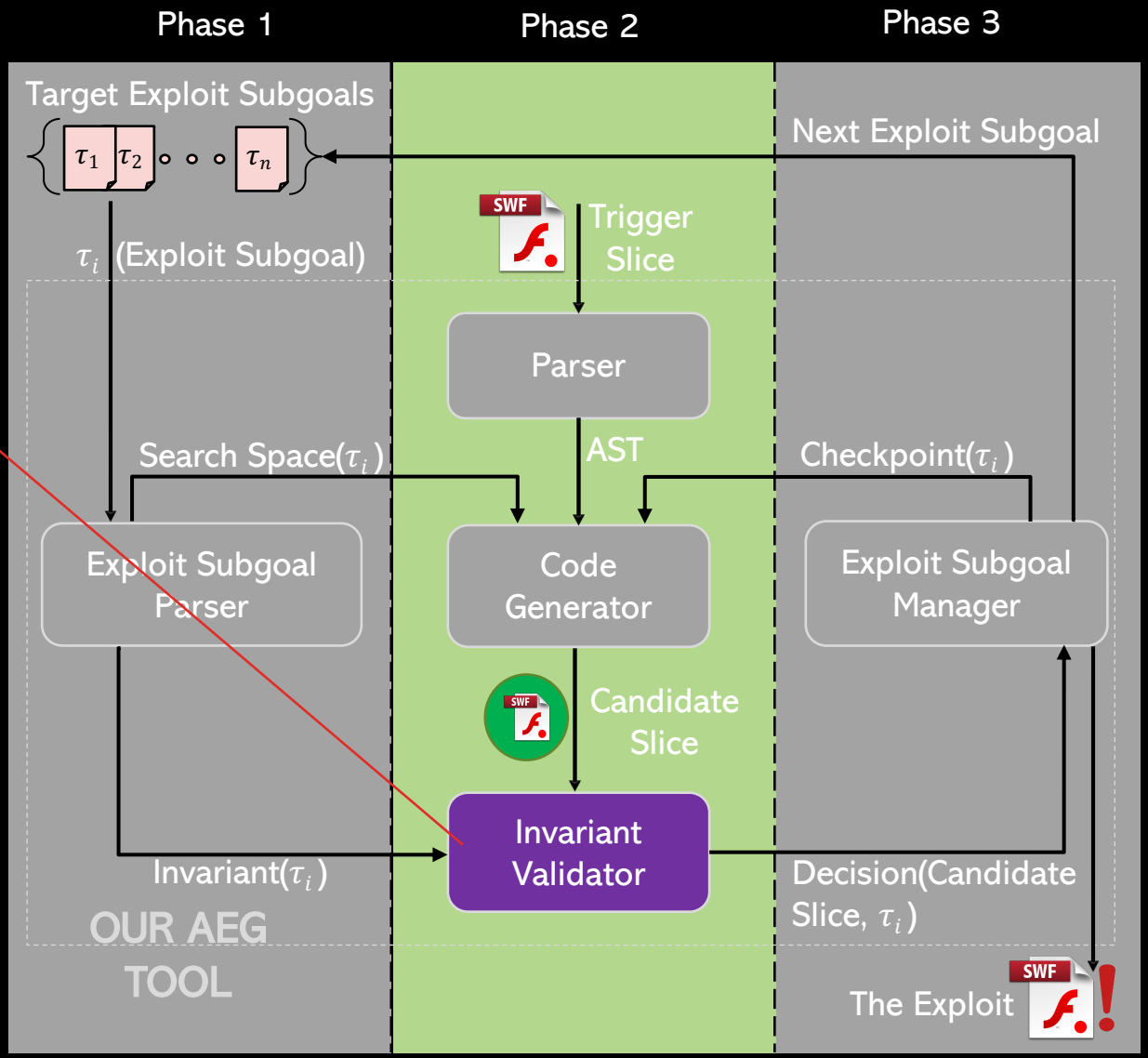
Generates candidate slices



# Phase 2: Generates Candidate Slices and Validating Invariant



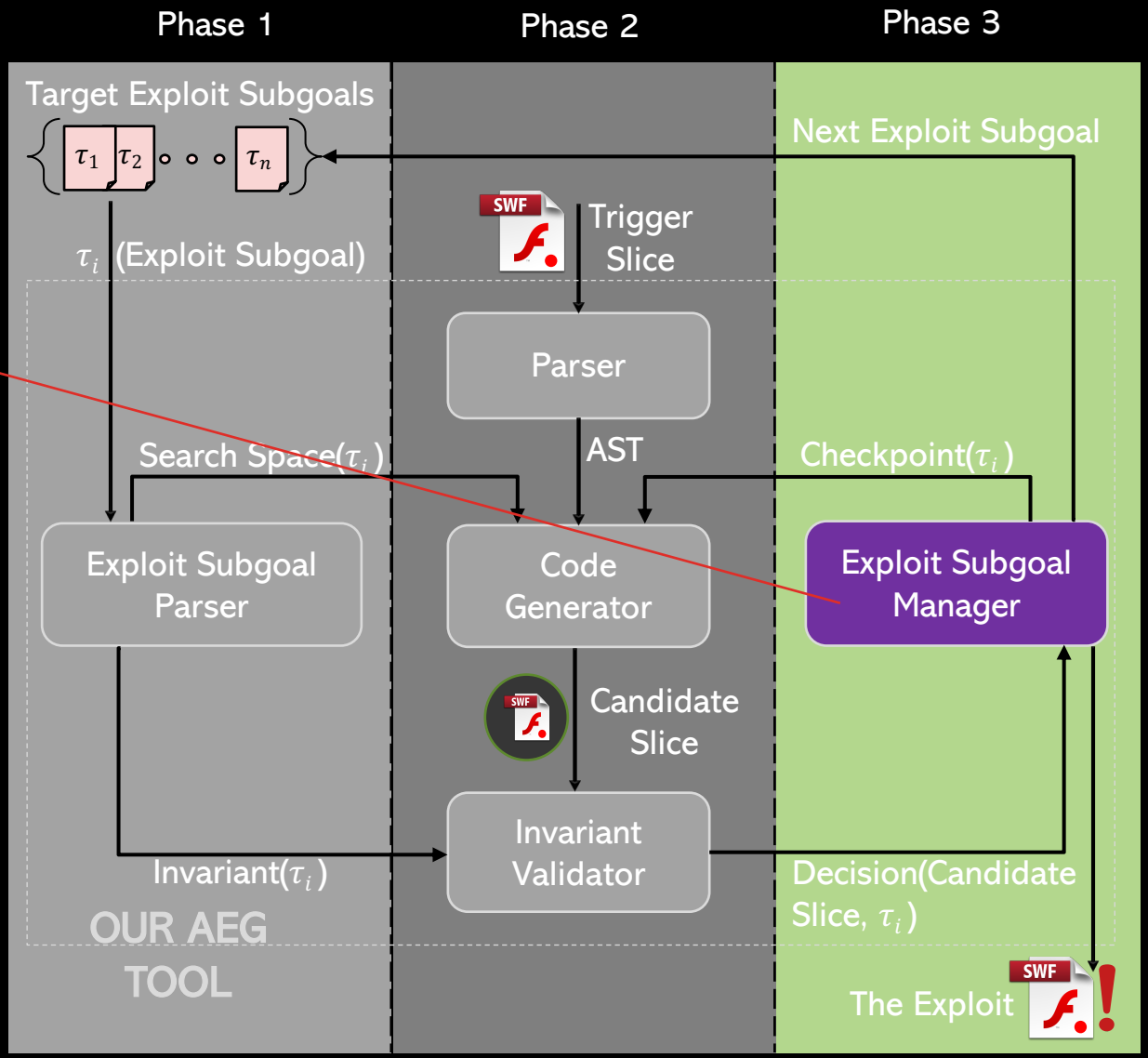
Executes and tests candidate slices



# Phase 3: Evaluating Candidate Slices

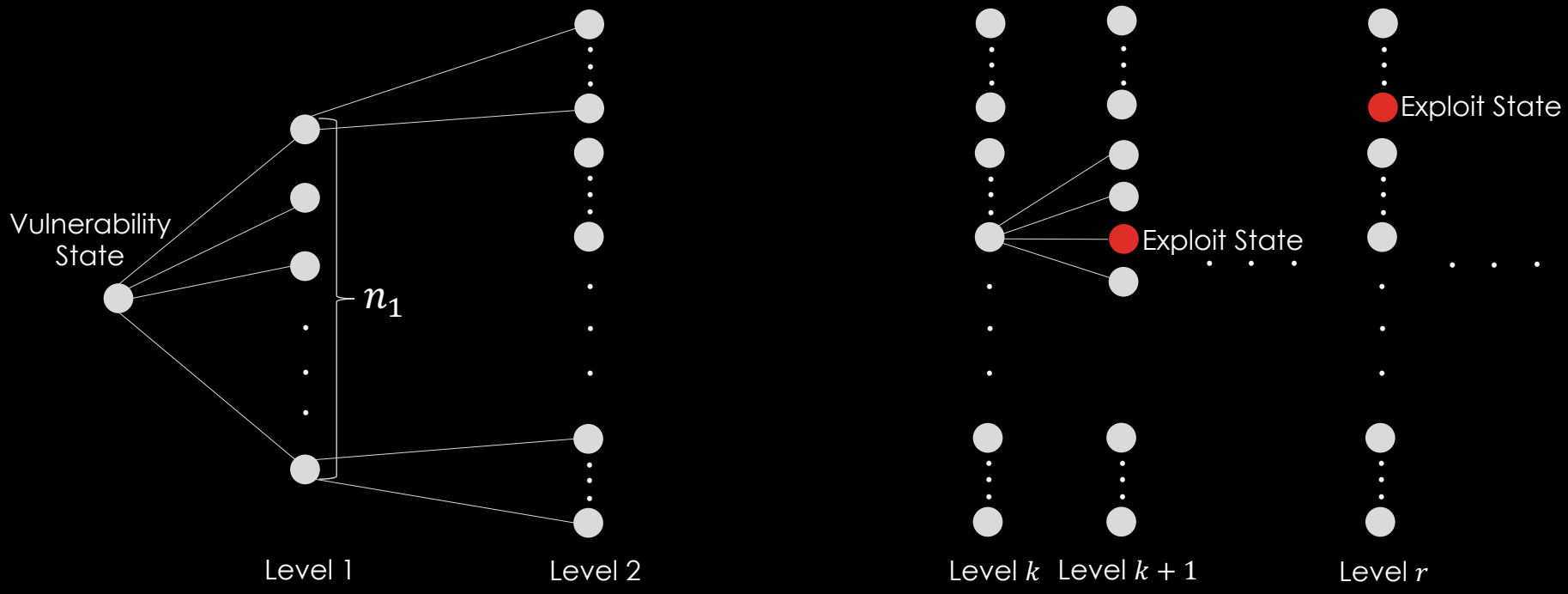


Evaluates the execution of candidate slices



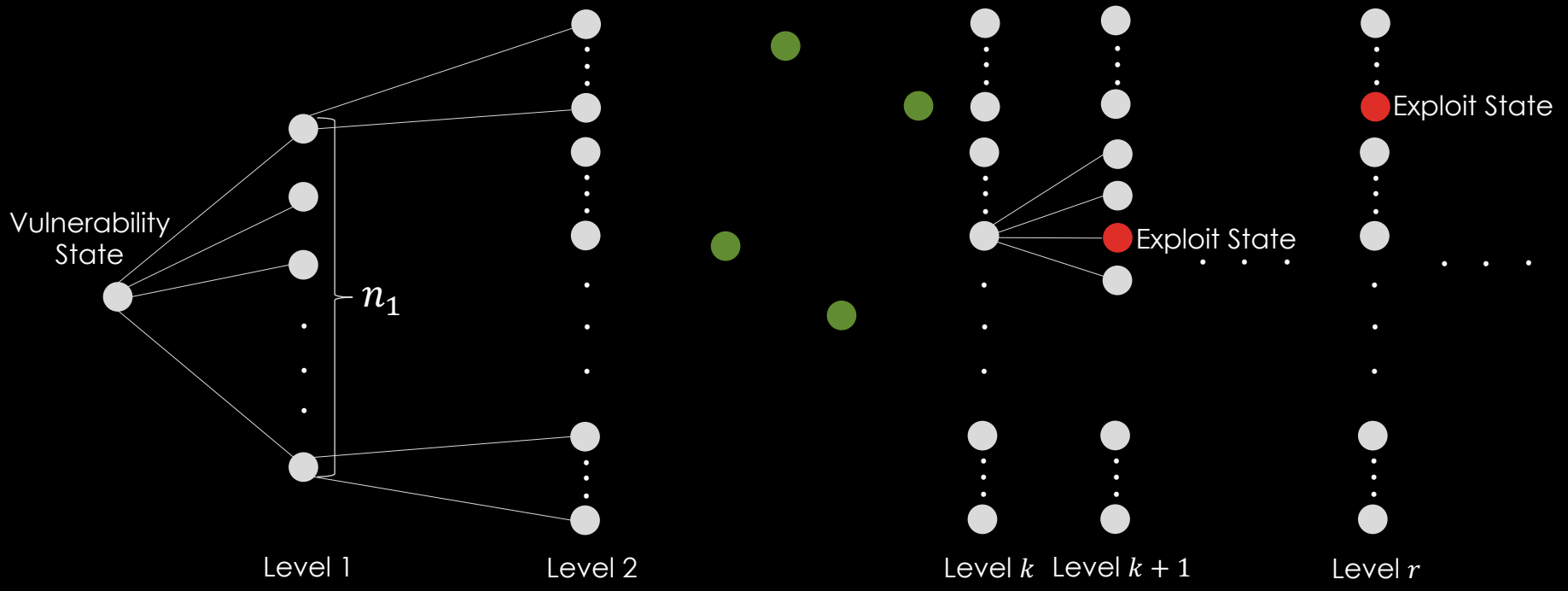
# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence



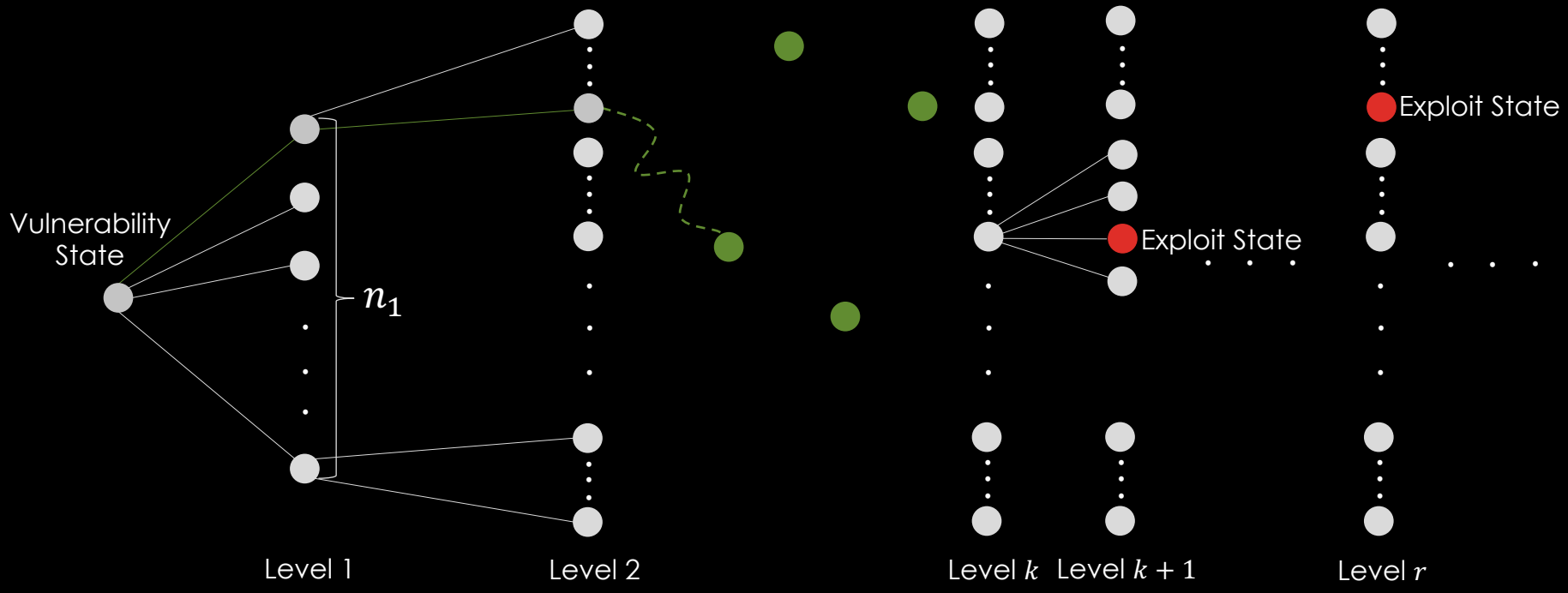
# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence



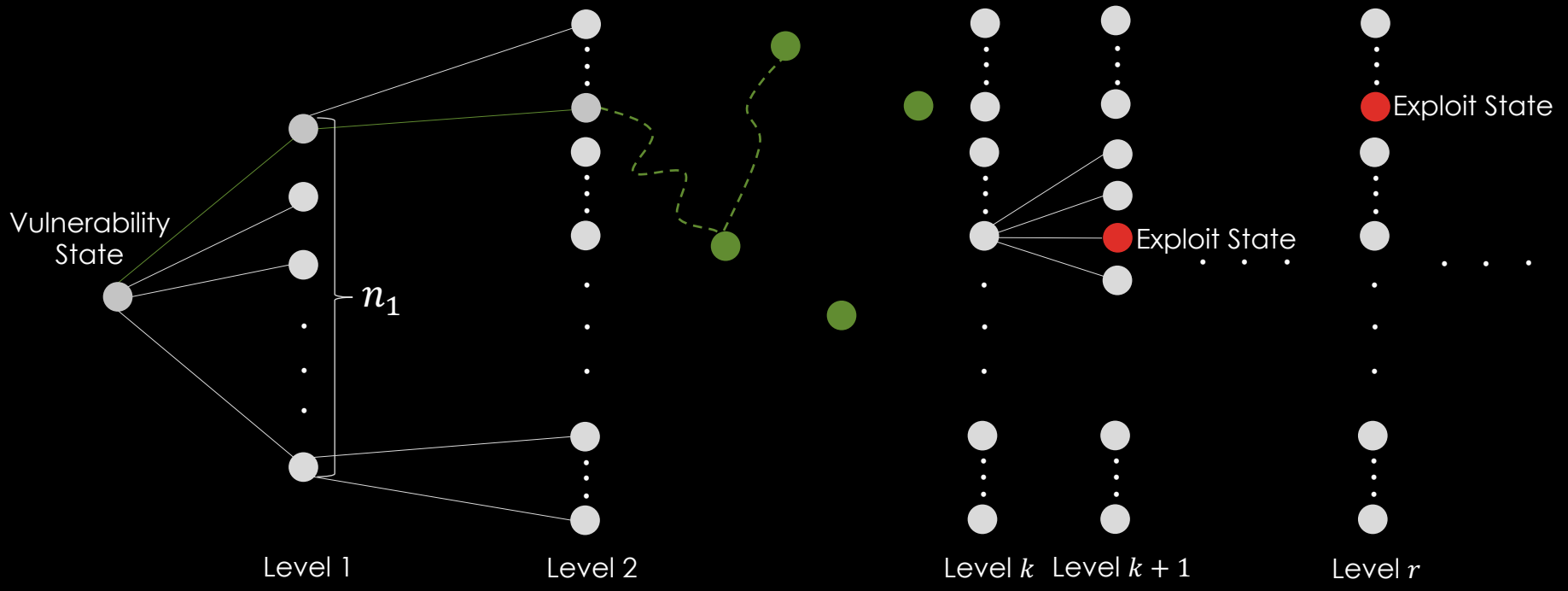
# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence



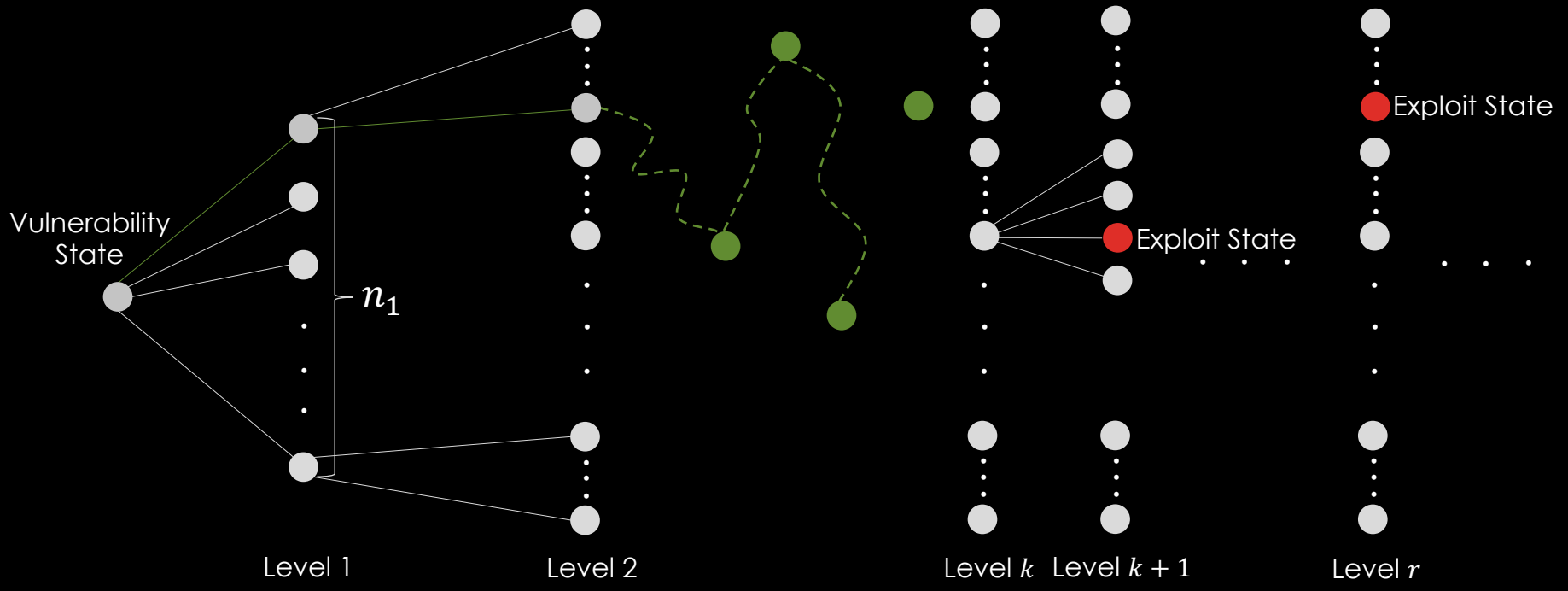
# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence



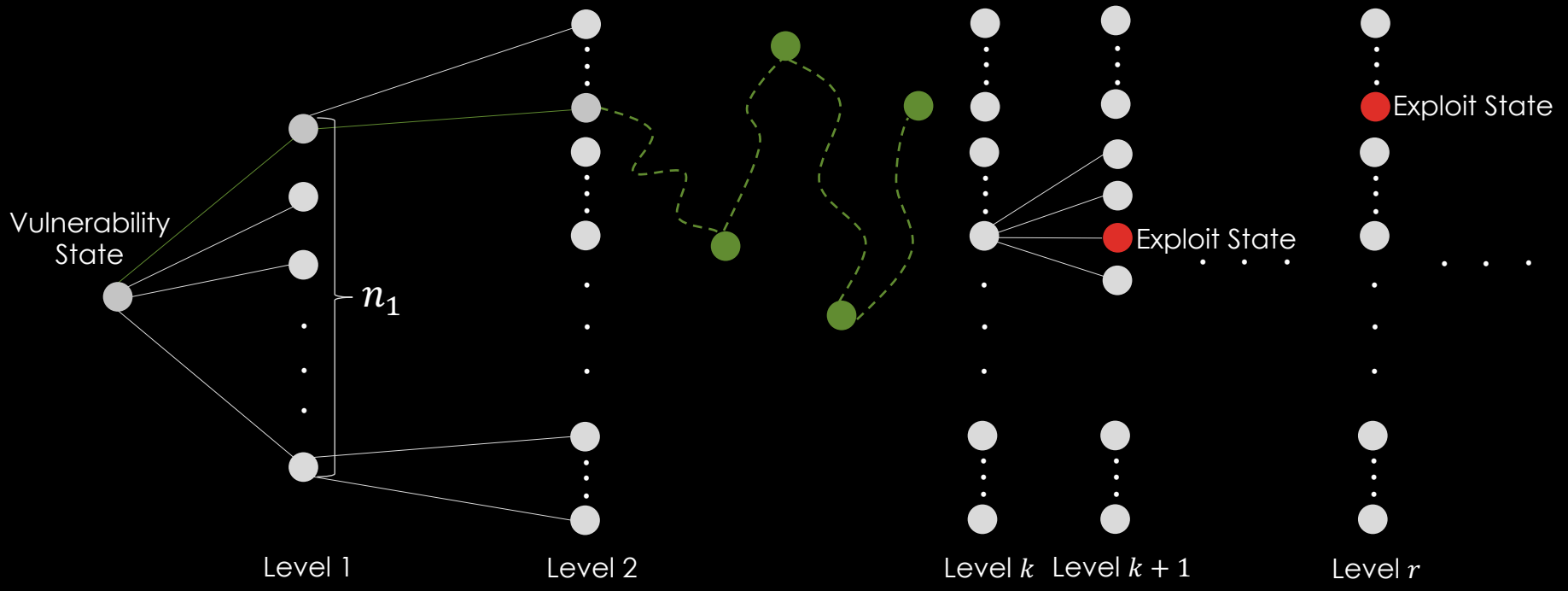
# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence



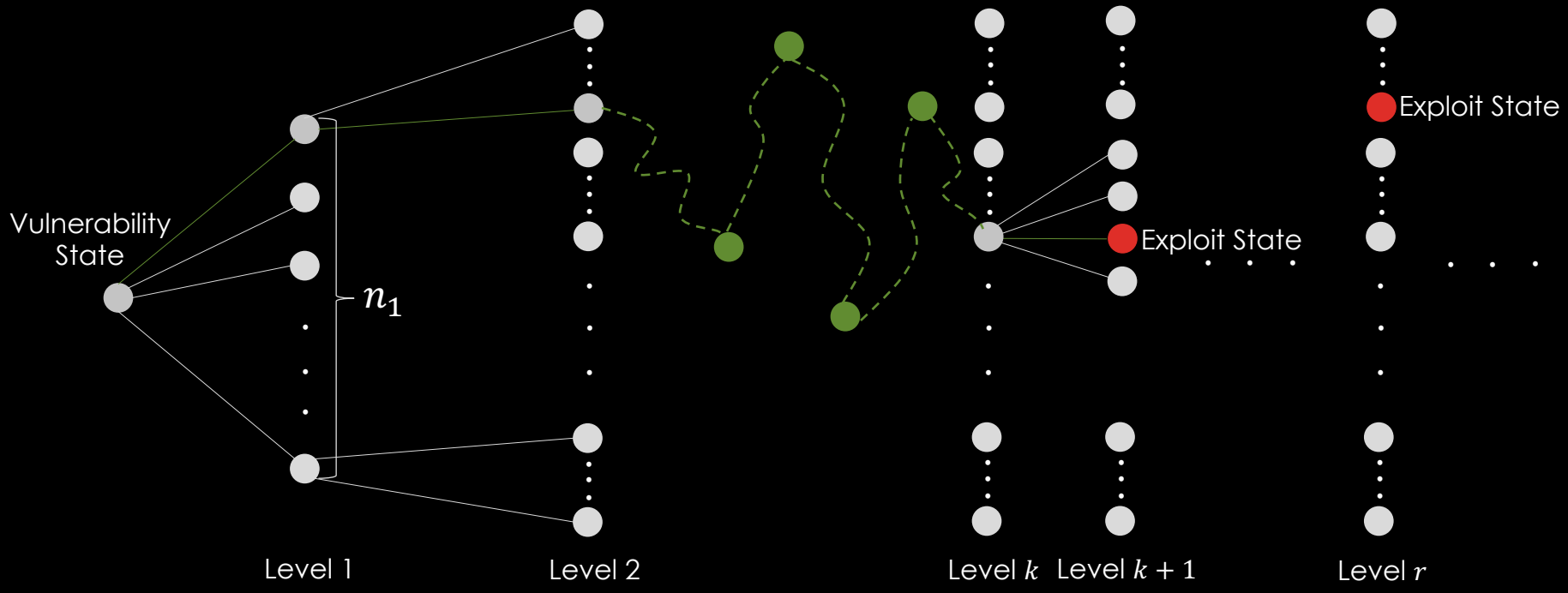
# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence



# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence



# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence
- Operand Stack Verification $(98.78\%)$
  - Disqualifies candidate slices that perform illegal stack operations

# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence
- Operand Stack Verification $(98.78\%)$
  - Disqualifies candidate slices that perform illegal stack operations
- Instruction Tiling $(\sim 10^{13.5})$
  - Glues instructions together to obtain more coarse-grained, meaningful instruction chains

# Optimization Techniques

- Exploit Deconstruction $(\sim 10^{45})$
  - Synthesizing smaller exploit subgoals in sequence
- Operand Stack Verification $(98.78\%)$
  - Disqualifies candidate slices that perform illegal stack operations
- Instruction Tiling $(\sim 10^{13.5})$
  - Glues instructions together to obtain more coarse-grained, meaningful instruction chains
- Feedback from the AVM $(58\%)$
  - Detects error-raising instruction prefixes

# Experimental Results -I

- Two sets of experiments
  - CVE-2015-5119
  - Open-source core implementation
  - Closed-source standalone Flash Player Debugger

| Exploit Subgoal | Number of Generated Candidate Slices | Number of Executed Candidate Slices | Percentage of Executed Candidate Slices | Synthesizing Time (s) |
|---|---|---|---|---|
| Corrupting a Buffer Space Implicitly | 2,396,744 | 12,229 | 0.51 | 9.35 |
| Spraying Helper Elements | 19,173,952 | 73,997 | 0.38 | 55.90 |
| Locating Sprayed Elements | 37,448 | 357 | 0.95 | 1.72 |
| Disclosing the Offset of the Located Elements | 55,345,757 | 282,392 | 0.51 | 138.26 |
| Corrupting the Disclosed Buffer | 4,793,488 | 21,591 | 0.45 | 17.03 |
| Locating `ELF` Object Files | 19,173,952 | 81,545 | 0.42 | 57.12 |
| Locating `libc` Libraries | 55,345,757 | 278,385 | 0.50 | 138.05 |
| Locating Executable Segment | 76,695,808 | 379,587 | 0.49 | 199.78 |
| Locating Gadgets and Building the ROP Chain | 435,848,049 | 1,648,451 | 0.37 | 240.92 |
| | | | | **Total:** 858.13 (14m 18.13s) |
| Corrupting a Buffer Space Implicitly | 2,396,744 | 29,167 | 1.21 | 605.58 |
| Spraying Helper Elements | 19,173,952 | 210,225 | 1.09 | 3,895.64 |
| Locating Sprayed Elements | 37,448 | 769 | 2.05 | 12.76 |
| Disclosing the Offset of the Located Elements | 55,345,757 | 508,339 | 0.91 | 6,845.86 |
| Corrupting the Disclosed Buffer | 4,793,488 | 41,342 | 0.86 | 963.86 |
| Locating `ELF` Object Files | 19,173,952 | 201,852 | 1.05 | 3,364.89 |
| Locating `libc` Libraries | 55,345,757 | 459,336 | 0.82 | 6,276.25 |
| Locating Executable Segment | 76,695,808 | 706,031 | 0.92 | 9,546.07 |
| Locating Gadgets and Building the ROP Chain | 435,848,049 | 2,954,400 | 0.67 | 11,512.47 |
| | | | | **Total:** 43,023.38 (11h 57m 03.38s) |

# Experimental Results -I

- The difference is due to starting/closing of the Flash Player
  - It takes 85ms on average, equivalent to 89% of the time

| Exploit Subgoal | Number of Generated Candidate Slices | Number of Executed Candidate Slices | Percentage of Executed Candidate Slices | Synthesizing Time (s) |
|---|---|---|---|---|
| Corrupting a Buffer Space Implicitly | 2,396,744 | 12,229 | 0.51 | 9.35 |
| Spraying Helper Elements | 19,173,952 | 73,997 | 0.38 | 55.90 |
| Locating Sprayed Elements | 37,448 | 357 | 0.95 | 1.72 |
| Disclosing the Offset of the Located Elements | 55,345,757 | 282,392 | 0.51 | 138.26 |
| Corrupting the Disclosed Buffer | 4,793,488 | 21,591 | 0.45 | 17.03 |
| Locating `ELF` Object Files | 19,173,952 | 81,545 | 0.42 | 57.12 |
| Locating `libc` Libraries | 55,345,757 | 278,385 | 0.50 | 138.05 |
| Locating Executable Segment | 76,695,808 | 379,587 | 0.49 | 199.78 |
| Locating Gadgets and Building the ROP Chain | 435,848,049 | 1,648,451 | 0.37 | 240.92 |
| | | | | **Total:** 858.13 (14m 18.13s) |
| Corrupting a Buffer Space Implicitly | 2,396,744 | 29,167 | 1.21 | 605.58 |
| Spraying Helper Elements | 19,173,952 | 210,225 | 1.09 | 3,895.64 |
| Locating Sprayed Elements | 37,448 | 769 | 2.05 | 12.76 |
| Disclosing the Offset of the Located Elements | 55,345,757 | 508,339 | 0.91 | 6,845.86 |
| Corrupting the Disclosed Buffer | 4,793,488 | 41,342 | 0.86 | 963.86 |
| Locating `ELF` Object Files | 19,173,952 | 201,852 | 1.05 | 3,364.89 |
| Locating `libc` Libraries | 55,345,757 | 459,336 | 0.82 | 6,276.25 |
| Locating Executable Segment | 76,695,808 | 706,031 | 0.92 | 9,546.07 |
| Locating Gadgets and Building the ROP Chain | 435,848,049 | 2,954,400 | 0.67 | 11,512.47 |
| | | | | **Total:** 43,023.38 (11h 57m 03.38s) |

# Experimental Results -II

- Generating exploit scripts for different vulnerabilities with the closed-source debugger

| Selected Vulnerabilities | Synthesizing Time | Flash Player Version |
|---|---|---|
| **CVE-2015-5119** | **11h 57m 03.38s** | **v11.2.202.262** |
| CVE-2013-0634 | 12h 09m 14.50s | v11.2.202.262 |
| CVE-2014-0502 | 12h 54m 15.19s | v11.2.202.262 |
| CVE-2014-0515 | 12h 51m 26.67s | v11.2.202.262 |
| CVE-2014-0556 | 12h 08m 35.29s | v11.2.202.262 |
| CVE-2015-0311 | 11h 56m 19.10s | v11.2.202.262 |
| CVE-2015-0313 | 12h 20m 47.98s | v11.2.202.442 |
| CVE-2015-0359 | 11h 05m 05.61s | v11.2.202.262 |
| CVE-2015-3090 | 12h 01m 33.16s | v11.2.202.262 |
| CVE-2015-3105 | 13h 25m 46.80s | v11.2.202.262 |
| CVE-2015-5122 | 12h 07m 02.59s | v11.2.202.262 |

# Limitations and Challenges

- Compatibility
  - Different memory offset
  - Calling external libraries
- Debuggers
  - PoCs perform their malicious activities *implicitly*
- Accuracy of the exploit subgoals
  - Having unnecessary instructions significantly increases the time (combinatorial rate)

# Thank You

**Fadi Yilmaz**
UNC Charlotte
*fyilmaz@uncc.edu*

**Meera Sridhar**
UNC Charlotte
*msridhar@uncc.edu*

**Wontae Choi**
*wtchoi.kr@gmail.com*

# Key References

[Younis et al. SQJ'16] Awad Younis, Yashwant K Malaiya, and Indrajit Ray. 2016. Assessing vulnerability exploitability risk using software properties. Software Quality Journal 24, 1 (2016), 159–202.

[Avgerinos et al. NDSS'11] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. 2011. AEG: Automatic Exploit Generation. In Proceedings of The Network and Distributed System Security Symposium (NDSS).

[Miller et al. ACM'90] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. Commun. ACM 33, 12 (1990), 32–44.

[Jayaraman et al. NFM'09] Karthick Jayaraman, David Harvison, and Adam Kiezun Vijay Ganesh. 2009. jFuzz: A concolic whitebox fuzzer for Java. In Proceedings of the First NASA Formal Methods Symposium (NFM).

[Rawat et al. NDSS'17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), Vol. 17. 1–14.