SAPIENZA
UNIVERSITÀ DI ROMA

Symbolic Execution

Emilio Coppa

SEASON Lab
season-lab.github.io

May 20, 2017

Why symbolic execution?

Real-world applications of this methodology:

- (Testing) bug detection
  e.g., inputs that crash an application
- (Security) exploit and backdoor identification
  e.g., inputs to bypass authentication code
- (Security) malware analysis
  e.g., generate vulnerability-based signatures
- (Security) reverse engineering of crypto code
  e.g., obtain an input that have generated an output

What is symbolic execution?

Originally introduced by James C. King in a 1976 paper as a static analysis technique for testing purposes.

Main idea:

- a symbol is associated to each variable, e.g., $i \mapsto \alpha_i$
- a symbol represents a set of input values, e.g., $\alpha_i \in [0, 2^{32} - 1]$
- conditional branches add *constraints* on symbols, restricting the input space of one or more symbols. If needed, execution is forked into into multiple (alternative) paths.
  E.g., `if i < 10` $\Rightarrow \alpha_i < 10$

Example: find inputs that lead to unsafe division

```
1.  void foobar(int a, int b) {
2.      int x = 1, y = 0;
3.      if (a != 0) {
4.          y = 3+x;
5.          if (b == 0)
6.              x = 2*(a+b);
7.      }
8.      assert(x-y != 0);
9.  }
```

Symbolic execution can find ***all*** inputs
that lead the assert to fail.

Example: symbolic execution

Execution state $(stmt, \sigma, \pi)$:

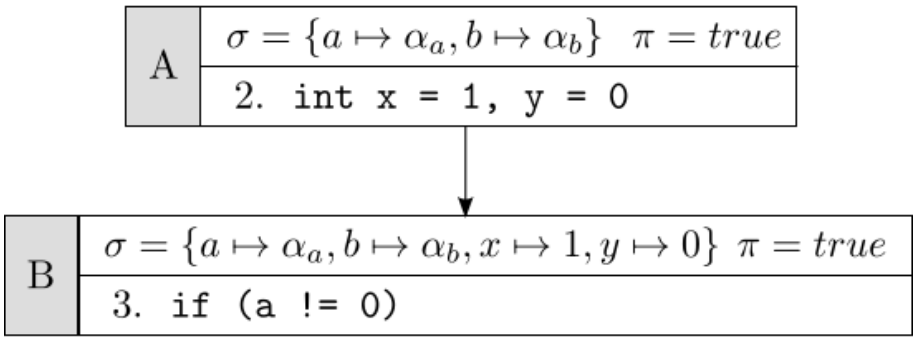
- *stmt* is the next statement to evaluate
- σ is a *symbolic store* that associates program variables with either expressions over concrete values or symbolic values α_i
- π denotes the path constraints, i.e., assumptions over α_i due to branches taken in the execution to reach *stmt*. Initially, $\pi = true$.

Initial state (first step):

A	$\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b\} \quad \pi = true$
	`2. int x = 1, y = 0`

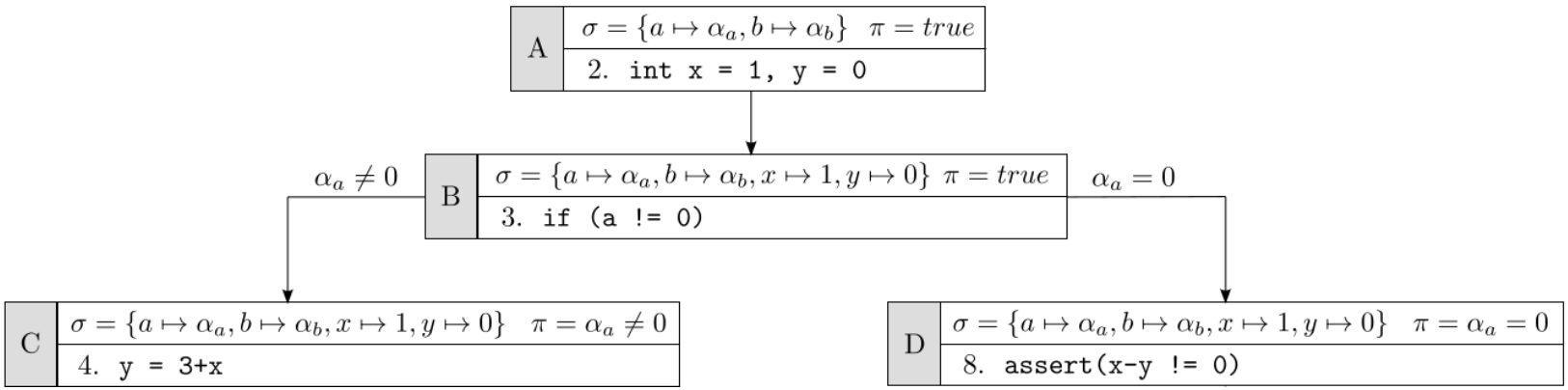
Example: symbolic execution

Second step:



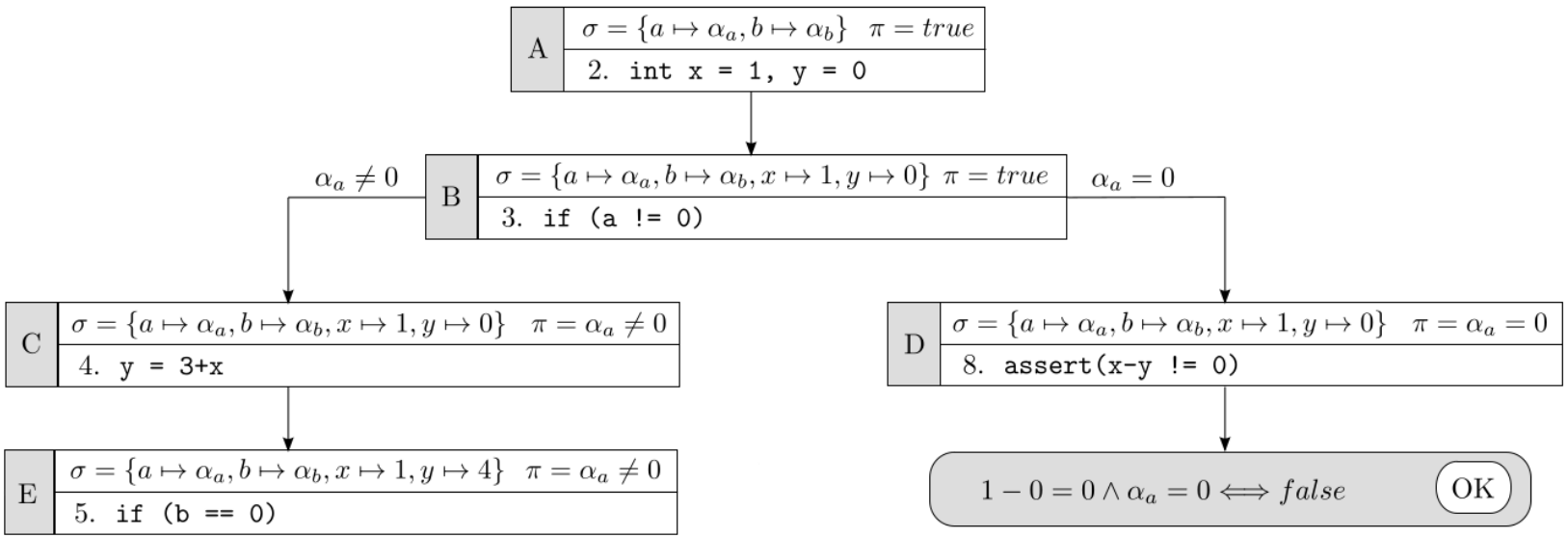
Example: symbolic execution

Third step:



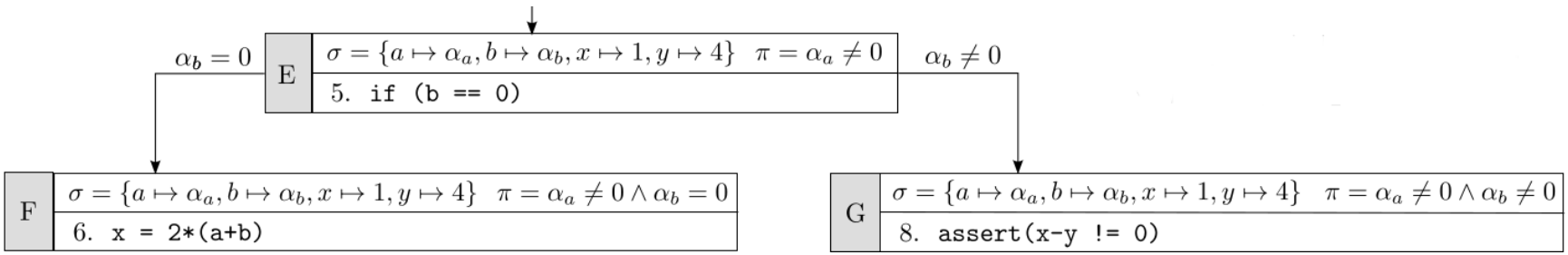
Example: symbolic execution

Fourth step:



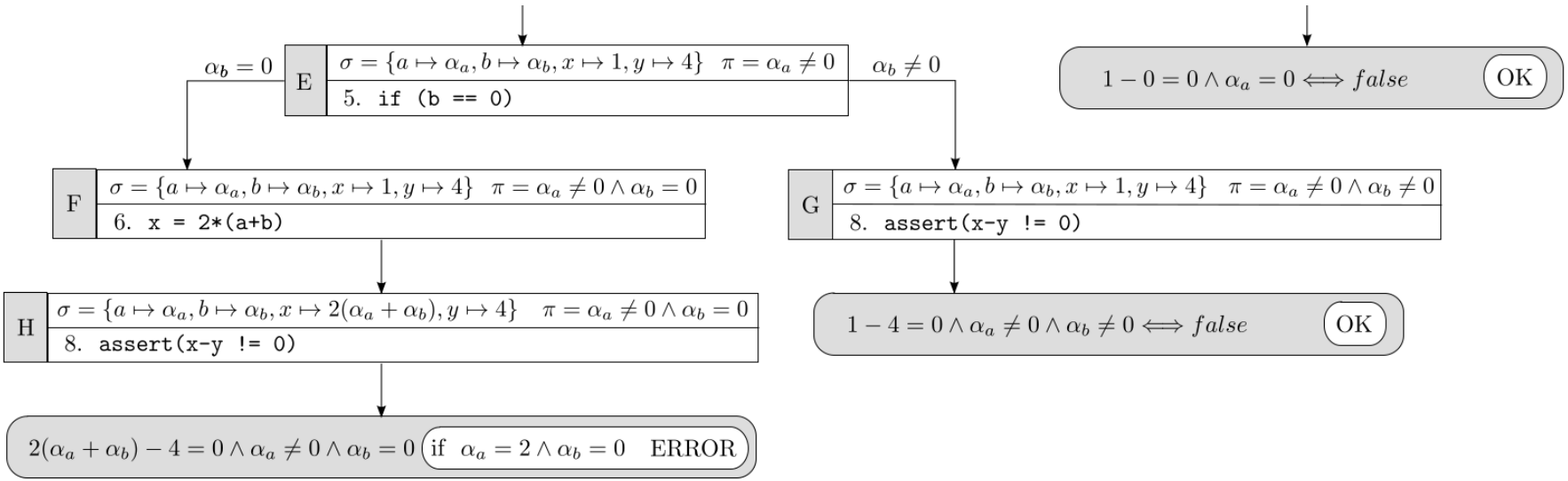
Example: symbolic execution

Fifth step:



Example: symbolic execution

Sixth step:



Example: discussion

Can we really prove that no crash is possible for that program?

Example: discussion

Can we really prove that no crash is possible for that program?

In "*theory*" yes, but in a more general setting:

1. constraints can be *hard* to solve
2. execution states to analyze can be many

Example: discussion

Can we really prove that no crash is possible for that program?

In "*theory*" yes, but in a more general setting:

1. constraints can be *hard* to solve
2. execution states to analyze can be many

Does symbolic execution scale to complex programs?

Symbolic execution: scalability issues?

"AEG: Automatic Exploit Generation" (T. Avgerinos et al.):

> *For example, KLEE is a state-of-the-art forward symbolic execution engine, but in practice is limited to small programs such as* /bin/ls.

Symbolic execution: scalability issues?

"AEG: Automatic Exploit Generation" (T. Avgerinos et al.):

> *For example, KLEE is a state-of-the-art forward symbolic execution engine, but in practice is limited to small programs such as* /bin/ls.

Pure static symbolic execution hardly scales

Symbolic execution: SAGE-NDSS08

Impact of *imperfect* symbolic execution:

SAGE: Whitebox Fuzzing for Security Testing

SAGE has had a remarkable impact at Microsoft.

Patrice Godefroid, Michael Y. Levin, David Molnar, Microsoft

Most *ACM Queue* readers might think of "program verification research" as mostly theoretical with little impact on the world at large. Think again. If you are reading these lines on a PC running some form of Windows (like 93-plus percent of PC users—that is, more than a billion people), then you have been affected by this line of work—without knowing it, which is precisely the way we want it to be.

Symbolic execution: SAGE-NDSS08 (2)

IMPACT OF SAGE

Since 2007, SAGE has discovered many security-related bugs in many large Microsoft applications, including image processors, media players, file decoders, and document parsers. Notably, SAGE found roughly *one-third* of all the bugs discovered by file fuzzing during the development of Microsoft's Windows 7. Because SAGE is typically run last, those bugs were missed by everything else, including static program analysis and blackbox fuzzing.

Finding all these bugs has saved millions of dollars to Microsoft, as well as to the world in time and energy, by avoiding expensive security patches to more than 1 billion PCs. The software running on *your* PC has been affected by SAGE.

Since 2008, SAGE has been running 24/7 on an average of 100-plus machines/cores automatically fuzzing hundreds of applications in Microsoft security testing labs. This is more than 300 machine-years and the *largest computational usage ever for any SMT (Satisfiability Modulo Theories) solver*, with more than 1 billion constraints processed to date.

SAGE is so effective at finding bugs that, for the first time, we faced "bug triage" issues with dynamic test generation. We believe this effectiveness comes from being able to fuzz large applications (not just small units as previously done with dynamic test generation), which in turn allows us to find bugs resulting from problems across multiple components. SAGE is also easy to deploy, thanks to x86 binary analysis, and it is fully automatic. SAGE is now used daily in various groups at Microsoft.

Symbolic execution: many problems to address

A symbolic engine is built on top of several kinds of ingredients:

1. *memory model*
2. *path explosion*
3. *state scheduling*
4. *symbolic executors*
5. *interaction with environment*
6. *constraint solvers*

Each solution chooses a set of techniques and tools

- many trade-offs to consider (e.g., completeness vs. efficiency)
- different choices for different goals

Symbolic execution: memory model

How do we hanlde *symbolic reads* or *symbolic writes*?

```
int array[10] = { 0 };
array[i] = 10;
assert(array[j] != 0);
```

Different trade-offs

- *fully symbolic memory*: e.g., fork state for each possible value of a symbol
- *address concretization*: e.g., evaluate state considering maximum value of a symbol
- *partial memory modeling*: e.g., allow symbolic reads, concretize symbolic writes

Memory model has a huge impact!

Symbolic execution: path explosion

Each undecidable branch forks execution: exponential explosion of paths. Preconditioned symbolic execution ([AEG-NDSS11]) deals with this aspect by imposing preconditions on the input space:

- *None*
- *Known Length*:
  e.g., interesting inputs have a minimum length
- *Known Prefix*:
  e.g., interesting inputs have a known prefix
- *Concolic Execution*:
  consider only a single (concrete) value for an input

By limiting the size of input space, symbolic execution can explore the target program more efficiently. If a precondition is too specific, no bugs or exploits will be found.

Symbolic execution: preconditions (example)

```
// buffer of size S
// N symbolic branches
if (input[0] < 42) [...]
[...]
if (input[N - 1] < 42) [...]

// symbolic loop
strcpy(dest, input); // a loop
```

Impact of preconditions on state space:

- *None*: $2^N \cdot S$.
- *Known Length*: constraint that $(S-1)$ bytes of `input` are not equal to \0. Loop length is known: 2^N.
- *Known Prefix*: first P bytes of `input` are known ($P < N < S$). $2^{N-P} \cdot S$. E.g., fixed header string.
- *Concolic Execution*: single path.

Symbolic execution: concolic execution

Concolic execution: mixed static/dynamic technique. Concrete values of some symbols are taken from a concrete (dynamic) execution and used in the symbolic (static) execution.

Goals:

- use faster dynamic techniques to test code, switch temporarily to symbolic execution to make progress
- use concrete values when reasoning on some constraints is too hard. Something is still better than nothing.
- (i) test program on a concrete input, (ii) track branches (taken flag + constraints), (iii) generate a new random input test that is compliant with constraints but explore new code

Symbolic execution: concolic execution (example)

```
void bar(int a, int b) {
    int x = foo(b);
    if (a == x) {
        // some critical error
    }
}
int foo(int v) {
  return (v * v) % 50; // complex constraint
}
```

Symbolic execution: concolic execution (example)

```
void bar(int a, int b) {
    int x = foo(b);
    if (a == x) {
        // some critical error
    }
}
int foo(int v) {
  return (v * v) % 50; // complex constraint
}
```

A pure symbolic approach may be unable to explore error path!

Symbolic execution: concolic execution (example)

```
void bar(int a, int b) {
    int x = foo(b);
    if (a == x) {
        // some critical error
    }
}
int foo(int v) {
  return (v * v) % 50; // complex constraint
}
```

A pure symbolic approach may be unable to explore error path!

Concolic approach

Generate random values for `a` and `b`. Execute on these inputs but track constraints. Negate constraints on the branch in order to generate a new inputs that explore error path.

Symbolic execution: path explosion (2)

Static or dynamic techniques can be used to limit state space.

E.g.,:

- *program slicing*: computation of the set of programs statements, the program slice, that may affect the values at some point of interest
- *taint-analysis*: method for tracking which variables can be modified by the user input

Symbolic execution: symbolic executors

Keeping in memory all the execution states is expensive.

Executor strategies:

- *offline executors*: one path at time, every run independent from the others, exploits concrete execution. E.g., [SAGE-NDSS08]
- *online executors*: for each fork, the execution state is cloned (COW). All active execution states are kept in memory. Need to guarantee isolation. E.g., [KLEE-OSDI08, CKC-TOCS12, AEG-NDSS11].
- *hybrid executors*: mixed approach, use of checkpoints. E.g., [MAYHEM-SP12].

Critical aspect w.r.t. scalability

Symbolic execution: state scheduling

Many path to explore: need to give priority to some paths.

Many heuristics:

- *depth first search*: many papers, minimize memory overhead
- *random path selection*: many papers, avoid starvation
- *coverage optimize search*: e.g., [KLEE-OSDI08]
- *buggy-path-first*: e.g., [AEG-NDSS11]
- *loop exhaustion*: e.g., [AEG-NDSS11]
- *pointer-detection*: e.g., [MAYHEM-SP12]
- ...

Papers adopt a mix of different heuristics
in order to reach different goals.

Symbolic execution: interaction with environment

Environment is an input source (e.g., syscalls). Hard to consider *all* possible outcomes of an interaction.

Emulation of the environment:

- *symbolic file system*: e.g., [KLEE-OSDI08]
- *symbolic sockets*: e.g., [AEG-NDSS11]
- *libraries*: e.g., [AEG-NDSS11]
- ...

Side-effects are approximated. This can significantly affect completeness of symbolic execution.

Symbolic execution: symbolic file system [KLEE-OSDI08]

Operations on:

- *concrete files*: actually performed
- *symbolic files*: emulated using a simple symbolic FS, private for each execution state. Users decide how many files and their size. Operation on unconstrained symbolic file generates $N + 1$ branches:
  - N: one for each symbolic file
  - $+1$: failing scenario

Emulation is done at library level to keep symbolic execution doable.

Symbolic execution: solvers

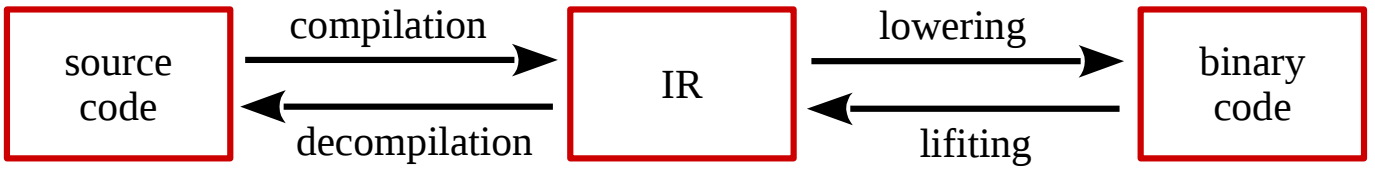
Some expressions of constraints are hard to solve (e.g., non linear constraints).

Some constraint solvers:

- STP [STP-TR07]: used by [EXE-CCS06, KLEE-OSDI08, MineSweeper-BOTNET08]
- Z3 [Z3-TACS08]: used by [FIRMALICE-NDSS15, MAYHEM-SP12]
- Dissolver [DISSOLVER-TR03]: initially used by [SAGE-NDSS08]
- Parma Polyhedra Library [PPL-SCP08]: used by [AEG-NDSS11]

Symbolic execution: source code vs binary code

Some papers target source code, others target binary code:



Examples:

- [KLEE-OSDI08] works on LLVM bytecode (source code to IR)
- [FIRMALICE-NDSS15] works on VEX IR (binary code to IR)

Static analysis of source code can provide useful hints about input and code properties. These are often exploited by symbolic execution to explore only a subset of paths.

Symbolic execution: a survey

- A Survey of Symbolic Execution Techniques. Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, Irene Finocchi. `https://arxiv.org/abs/1610.00502`

Symbolic execution: few security-related examples

- Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. NDSS 2015.

  Symbolic execution is used to find inputs that bypass authentication code.

- Exploring Multiple Execution Paths for Malware Analysis. IEEE SP 2007.

  Symbolic execution is used to find events that force malwares to show malicious behaviors.

Symbolic execution: few security-related examples (2)

- AEG: automatic exploit generation. NDSS 2011

  Symbolic execution is used to automatically generate an exploit (e.g., find the string input that will be executed as a shell code).

- Unleashing MAYHEM on Binary Code. IEEE SP 2012

  Given a tainted jump, symbolic execution is used to find if EIP can be modified to execute a malicious payload.

Conclusions

- Symbolic execution is a powerful technique
- hard to make it scalable in a general context
- ...still widely adopted in specific contexts
- need to make compromises
- many non trivial problems to take care of
- ...but fun things can be done with it :)

References

[AEG-NDSS11] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: automatic exploit generation. NDSS 2011.
[DISSOLVER-TR03] Y. Hamadi. Disolver: A distributed constraint solver. Technical report, 2003.
[EXE-CCS06] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. ACM CCS 2006.
[FIRMALICE-NDSS15] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. NDSS 2015.
[K-ACM76] J. C. King. Symbolic execution and program testing. Communication of ACM 1976.
[KLEE-OSDI08] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI 2008.
[MAYHEM-SP12] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. IEEE SP 2012.

References (2)

[MineSweeper-BOTNET08] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Botnet Detection: Countering the Largest Security Threat, chapter Automatically Identifying Trigger-based Behavior in Malware. Springer 2008.

[SAGE-NDSS08] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated white-box fuzz testing. NDSS 2008.

[Z3-TACS08] L. De Moura and N. BjÃÿrner. Z3: An efficient smt solver. TACASâĂŹ08/ETAPS 2008.

[KKM-USEC05] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. SSYM 2005.

[DART-PLDI05] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. PLDI 2005.