Ed Clarke Symposium

David Brumley
Carnegie Mellon University

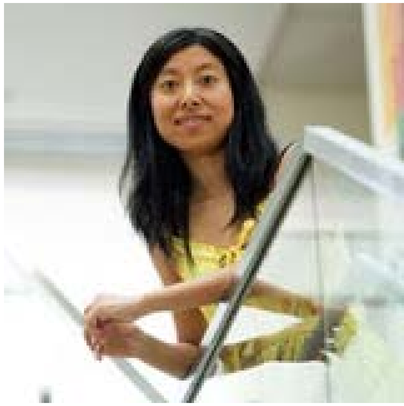
Dawn Song
UC Berkeley

Ed’s mentorship and help when I was a student, and later when I was a professor, has been invaluable.

Thank you.

Model Checking for Security Applications

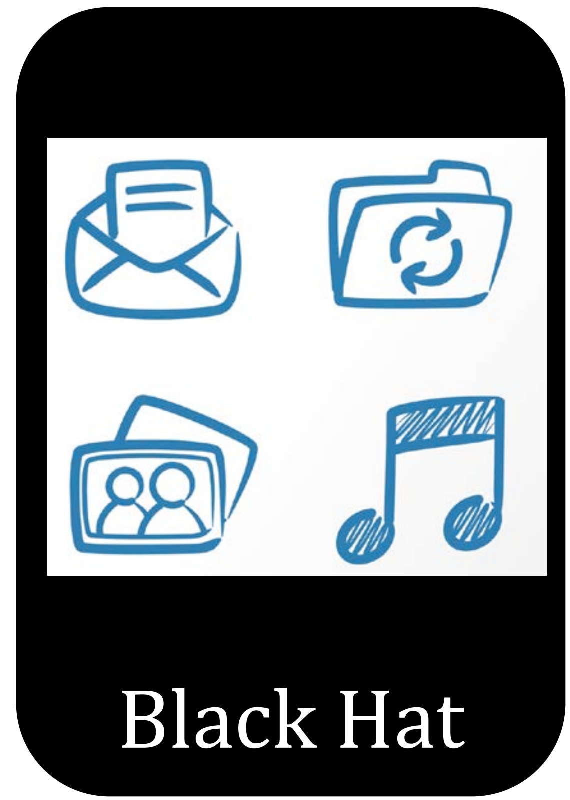
- Athena: an automatic checker for security protocol analysis
  - Work under Ed's mentorship



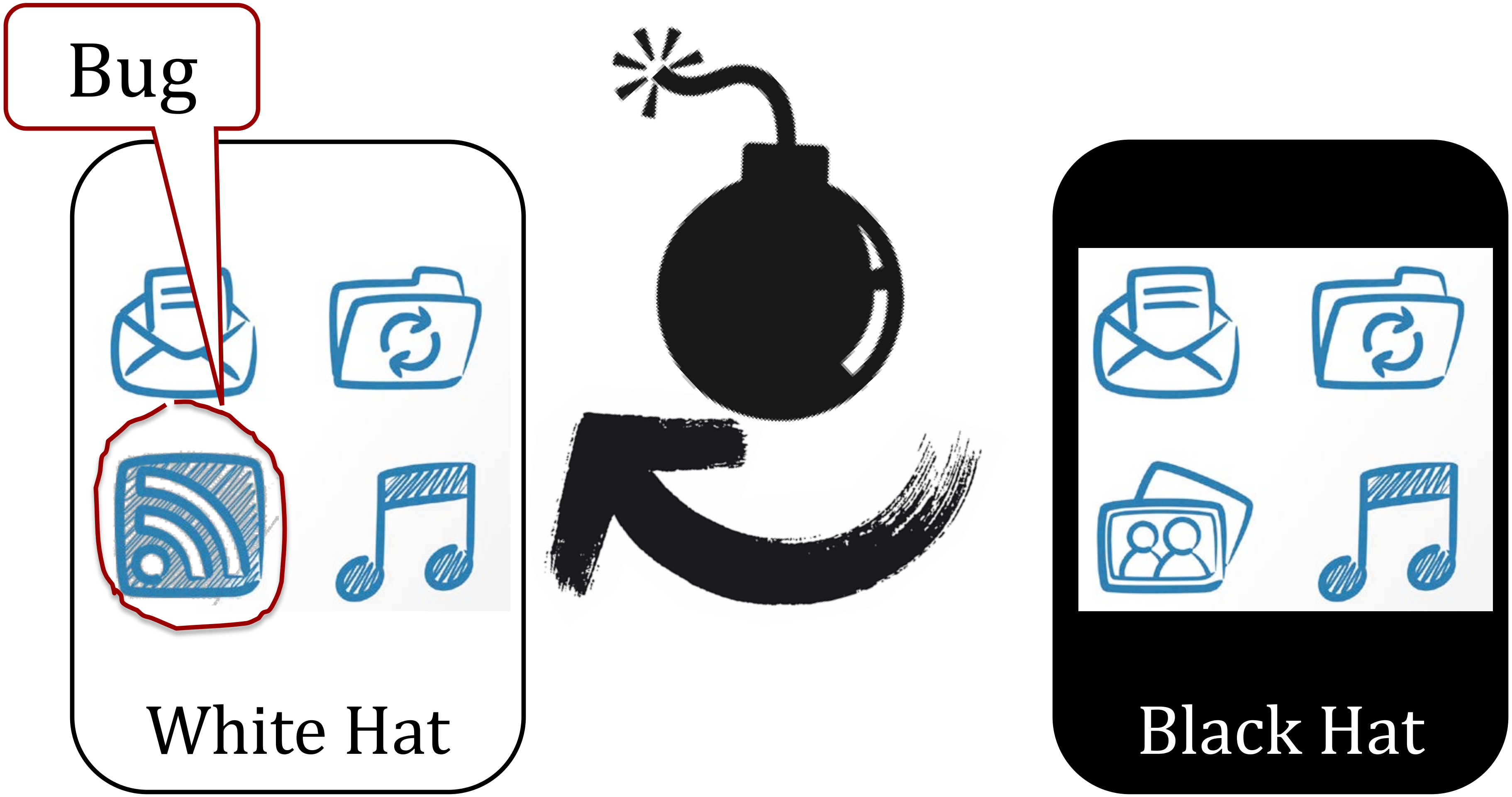
- BitBlaze: automatic security analysis of program binaries
  - E.g., Blitz: Compositional Bounded Model Checking for Real-World Programs
- WebBlaze: automatic security analysis and construction for web applications
  - E.g., first step towards building a formal foundation of web security

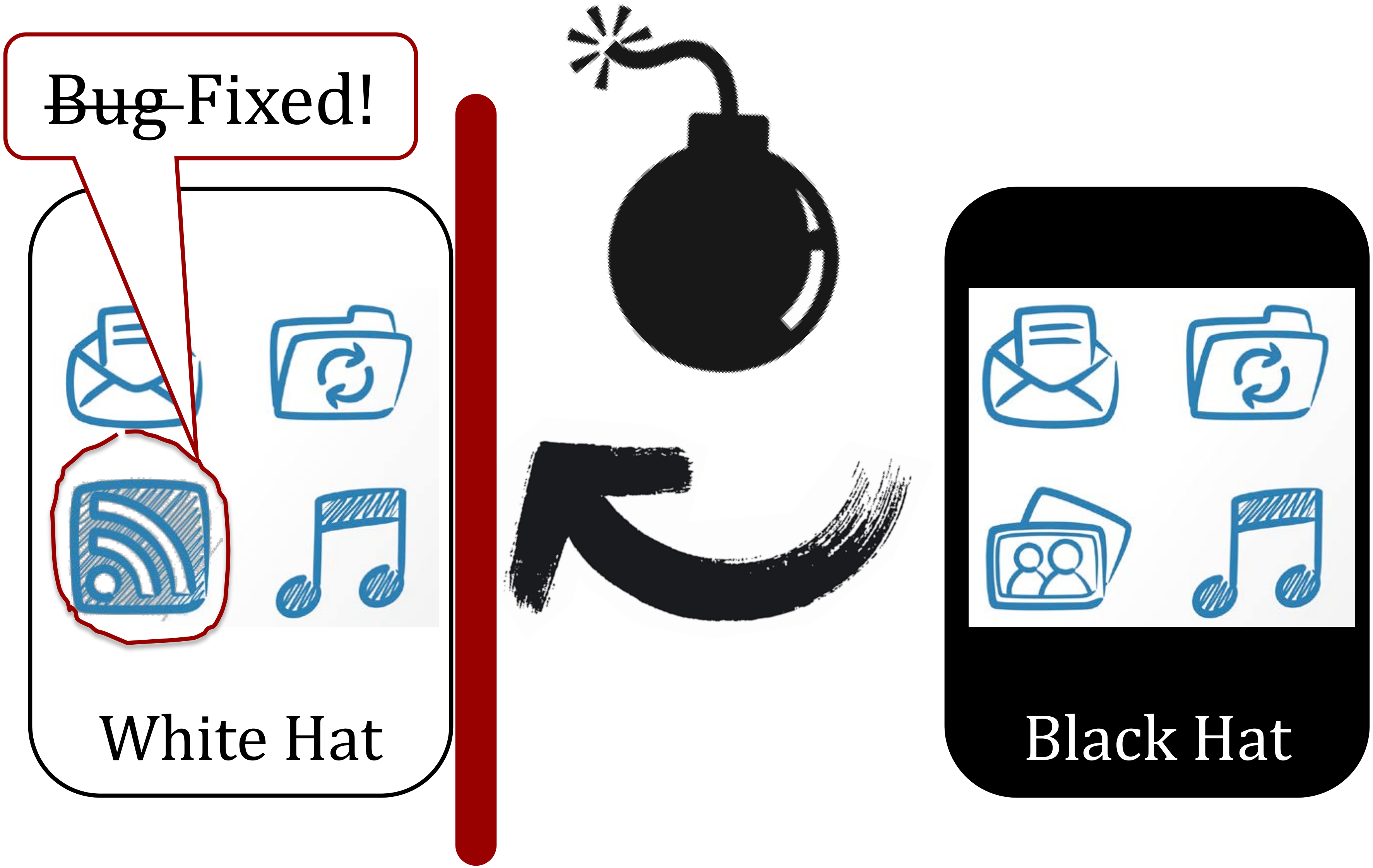
An epic battle

vs.



Exploit bugs

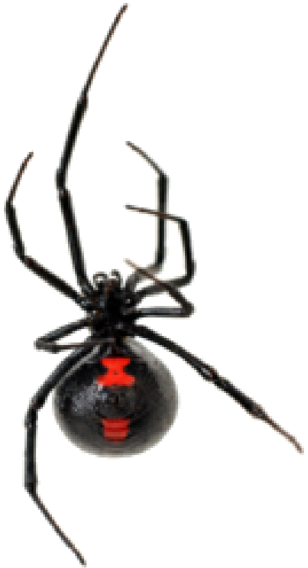




~~Bug~~ Fixed!
White Hat
Black Hat

Fact:

Windows, Mac, and Linux all have 100,000's of known bugs



Which bugs are exploitable?

Highly Trained Experts

Automatically
Check the World's
Software for
Exploitable Bugs

White

Inspiration

1.2 Advantages of Model Checking

Model Checking has a number of advantages compared to other verification techniques such as automated theorem proving or proof checking. A partial list of some of these advantages is given below:

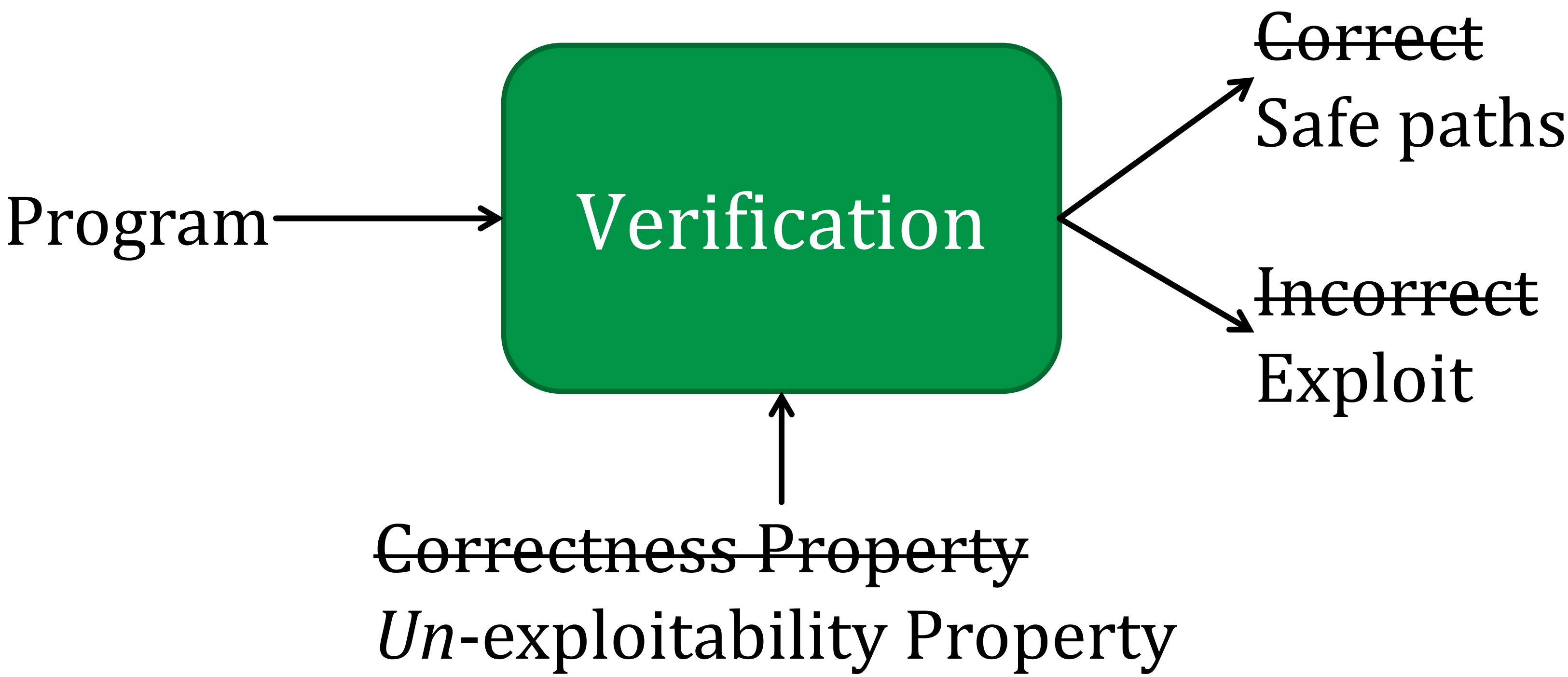
– No proofs! The user of a Model Checker d
rectness proof. In principle, all that is nece
description of the circuit or program to be
be checked and press the "return" key. T

– Fast. In practice, Model checking is fas
such as the use of a proof checker, which may require months of the user's time working in interactive mode.

– Diagnostic counterexamples. If the specification is not satisfied, the Model Checker will produce a counterexample execution trace that shows why the specification does not hold (Figure 2). It is impossible to overestimate the importance of the counterexample feature. The counterexamples are invaluable in debugging complex systems. Some people use Model Checking just for this feature.

If the property is a security property, the counter-example can be an exploit

Automated Exploit Generation[*]



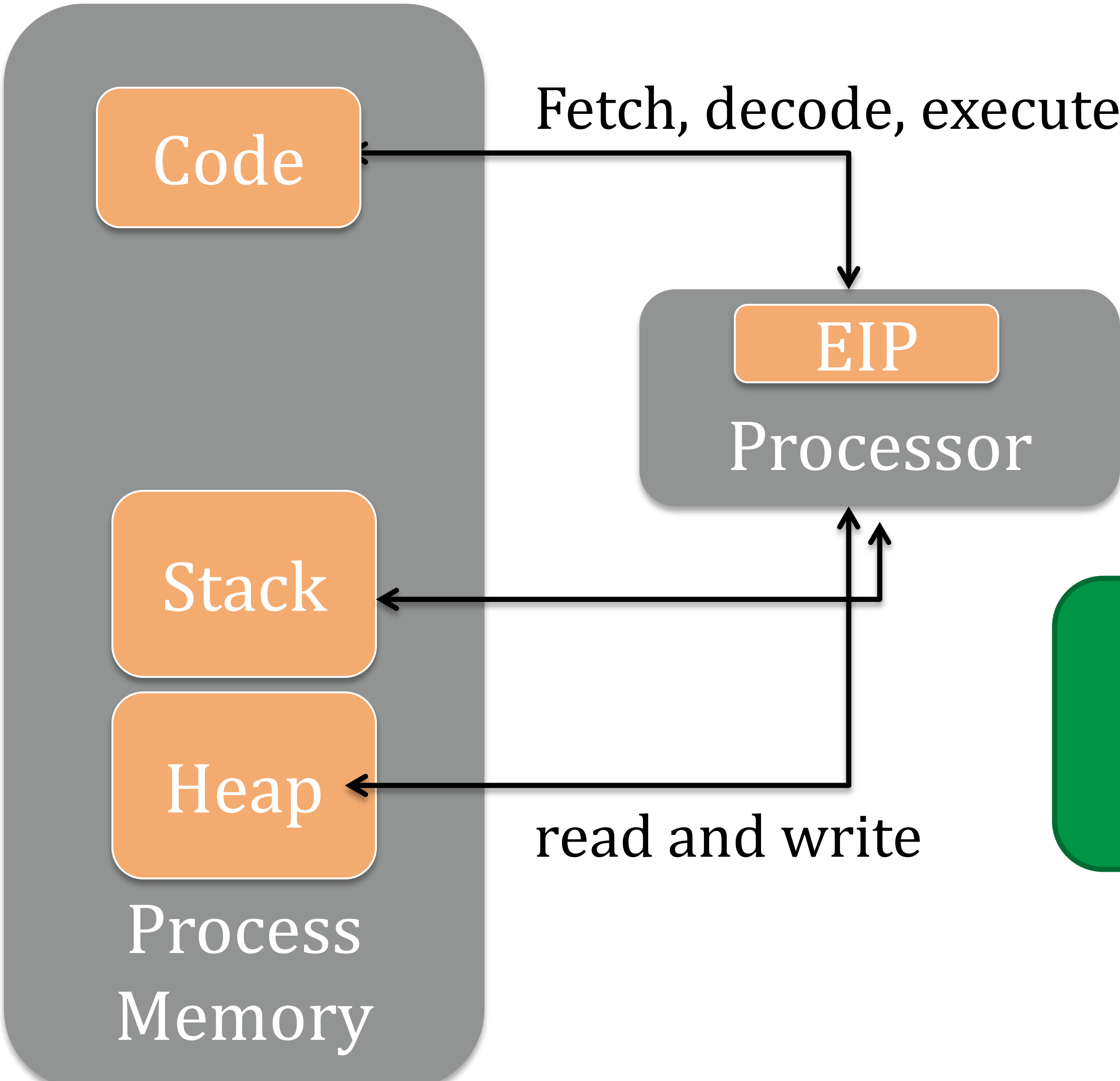
* Automatic Exploit Generation, NDSS 2011, CACM 2014

A brief history

2005	Automatic Discovery of API-Level Exploits [Ganapathy et al., Conference on Software Engineering]
2008	Automatic Patch-Based Exploit Generation [Brumley et al., IEEE Security and Privacy Symposium]
2010	Automatic Generation of Control Flow Hijack Exploits for Commodity Software [Heelan, MS Thesis]
2011	Automatic Exploit Generation [Avgerinos et al., Network and Distributed System Security Symposium]
2011	Q: Exploit Hardening Made Easy [Schwartz et al., USENIX Security Symposium]
2012	Unleashing Mayhem on Binary Code [Cha et al., IEEE Security and Privacy Symposium]

And >150 papers on symbolic execution

Basic Execution



<u>Un-exploitability</u>
Attackers cannot inject into EIP

checking Debian for exploitable bugs

37,000 programs

16 billion verification queries

~$0.28/bug
~$21/exploit

test cases

2,606,000 crashes

14,000 unique bugs

152 *new* exploits

* [ARCB, ICSE 2014, ACM Distinguished Paper], [ACRSWB, CACM 2014]

mining data

Q: How long do per-path queries take on average?

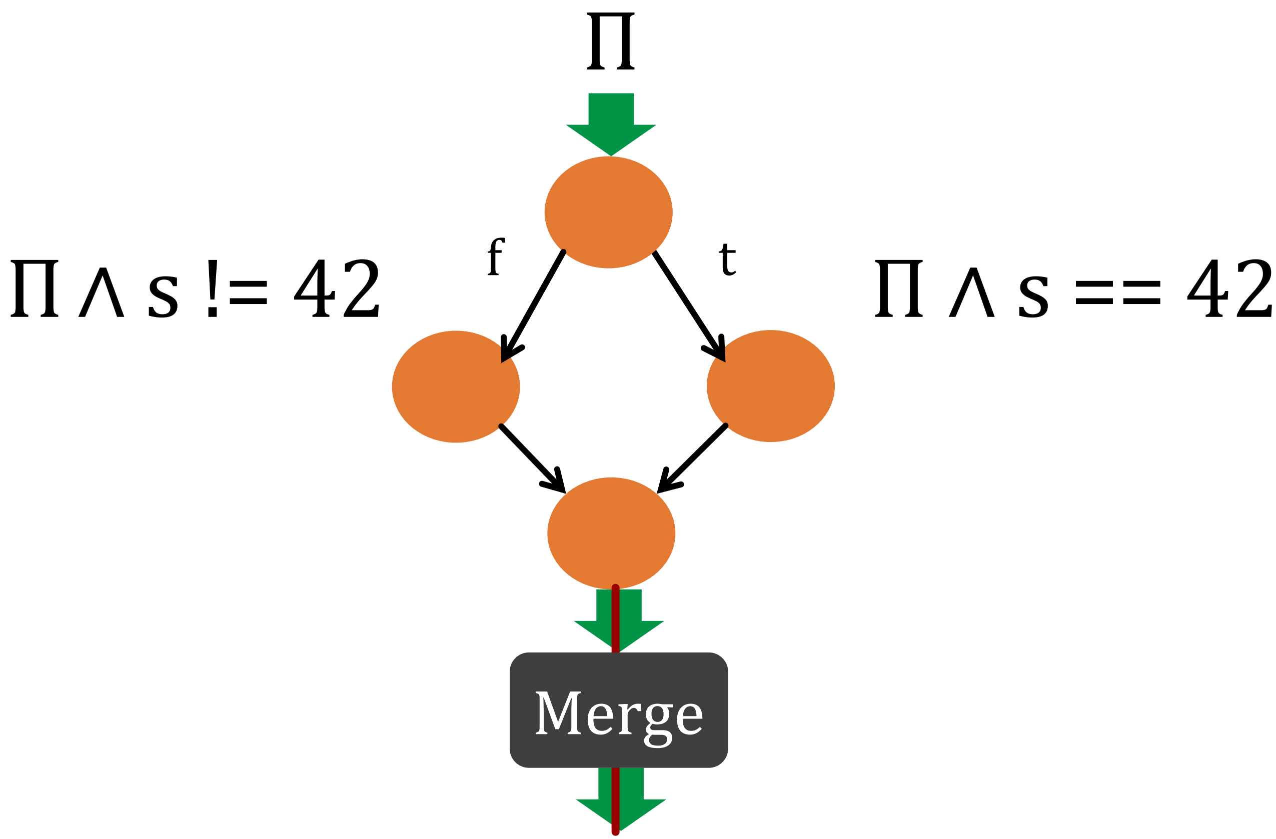
A: 3.67ms on average with 0.34 variance

Q: Should I optimize hard or easy formulae?

A: 99.99% take less than 1 second
and account for 78% of total time

optimize fast queries

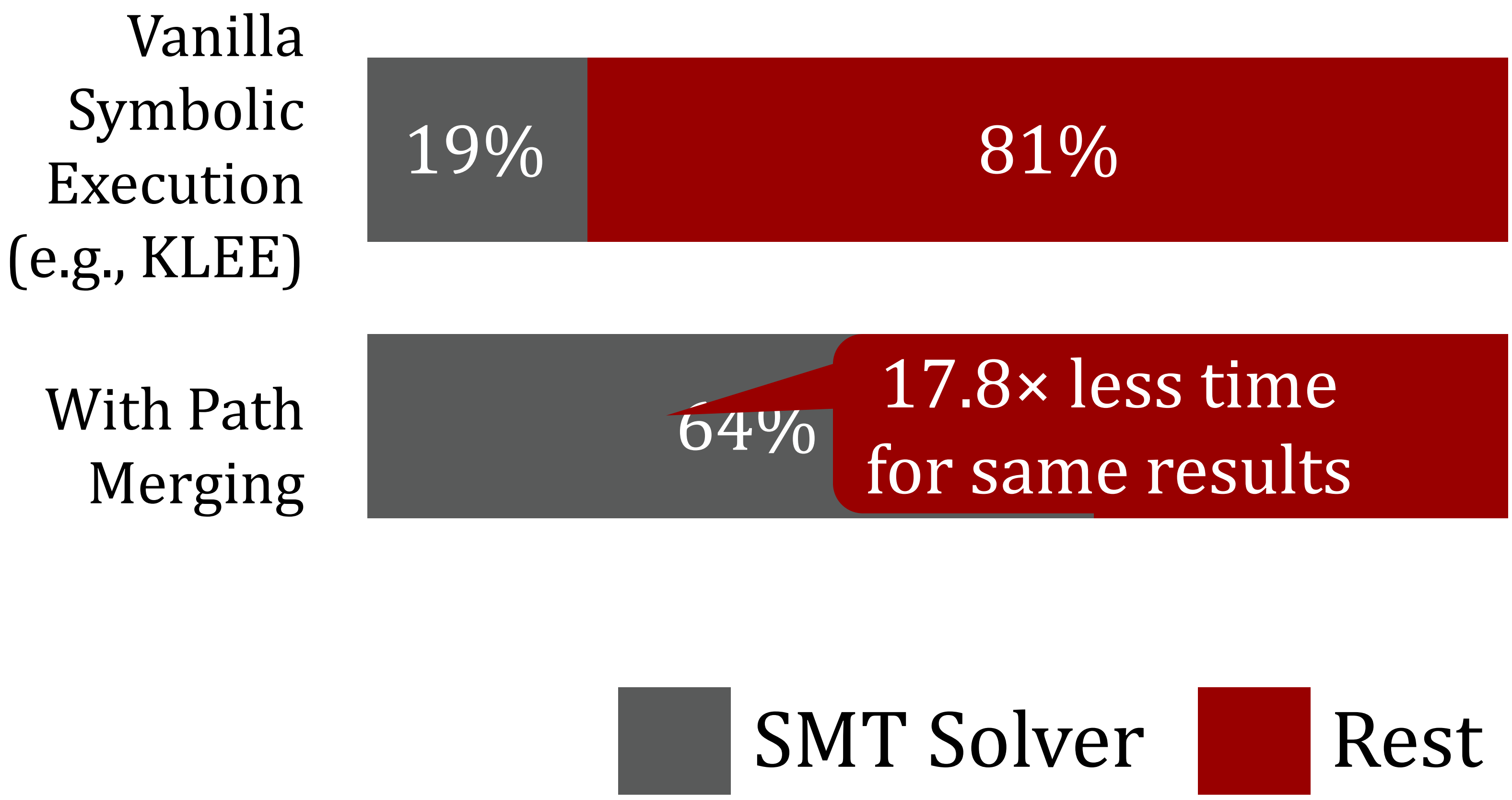
Path Merging[*]



$$\Pi' = (\Pi \land s \mathrel{!=} 42) \lor (\Pi \land s == 42)$$

* Veritesting, ICSE 2014

Execution Profile (Analysis Completes)



Vision:

Automatically Check the World's Software for *Exploitable* Bugs

What Hat

We're in the age of automated reasoning.
It seems wrong not to try.

Thank You Ed!

- David & Dawn

END