Targeted taint driven fuzzing using software metrics

Dustin Duran

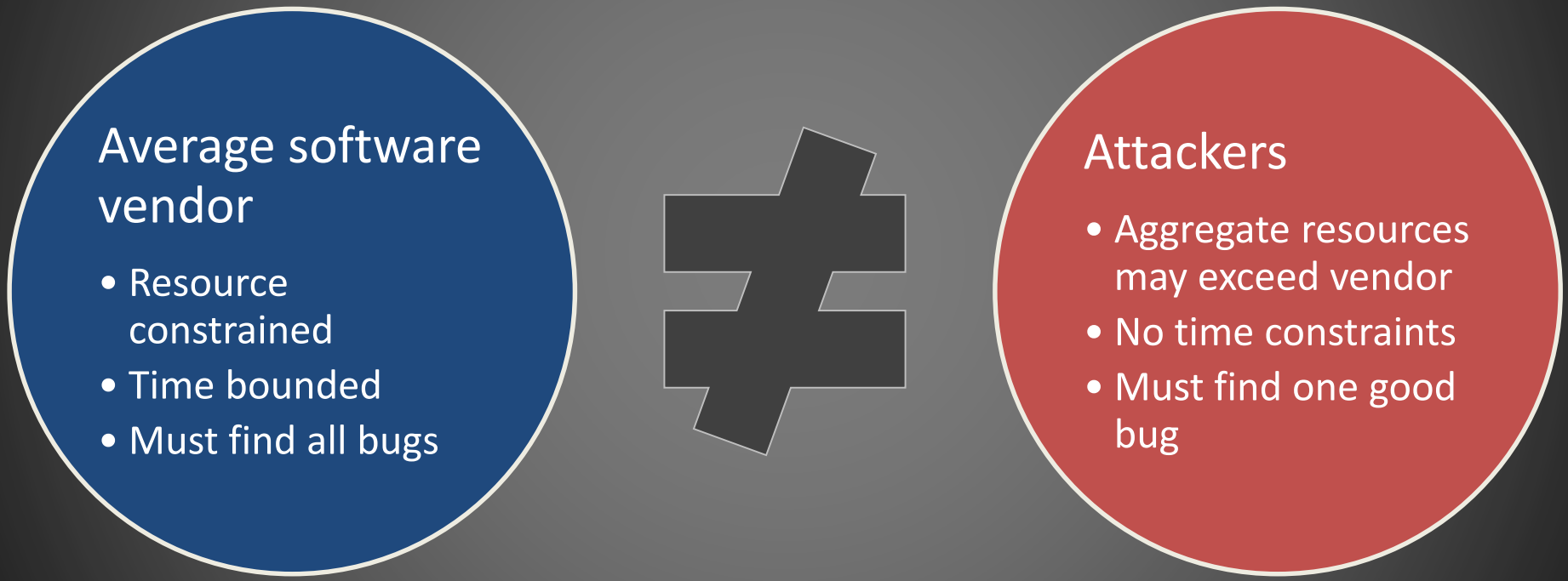
David Weston

Matt Miller

Microsoft

Framing the problem

The vulnerability finding imbalance



Software vendors must have superior technology or make significant resource/time investments

Fuzzing challenges

(for a software vendor)

- No single fuzzing engine finds all bugs
  - Dumb fuzzing is blind
  - Smart fuzzing is generally high cost & low ROI [Shirk08]
  - Constraint-based fuzzing is complex/heavyweight
  - ➢ Fuzzing innovations can provide vendors with a necessary edge
- Finite resources and time to devote to fuzzing
  - Tons of fuzzing happens at Microsoft, but still an upper bound
  - Which fuzzers are the best use of our time?
  - ➢ Optimizing overall effectiveness of fuzzing efforts is critical
- Fuzzing engine behavior is often opaque
  - What was covered (or NOT covered) during fuzzing?
  - Did the fuzzer hit the most concerning areas of code?
  - ➢ Deeper fuzzing insight improves confidence & aids gap analysis

Use dynamic taint analysis to select offsets for mutation

Dynamic trace	File Offset	Tainted Function
d.bmp.trace	d.bmp:7777	Func3
a.bmp.trace	a.bmp:4444	Func1
a.bmp.trace	a.bmp:8888	Func4
c.bmp.trace	c.bmp:6666	Func2
…	…	…

Taint driven fuzzing using software metrics

File Offset	Function	Cyclomatic Complexity
a.bmp:4444	Func1	40
a.bmp:5555	Func1	40
c.bmp:6666	Func2	37
d.bmp:7777	Func3	34
…	…	…

Use software metrics to prioritize the mutation order of the selected offsets

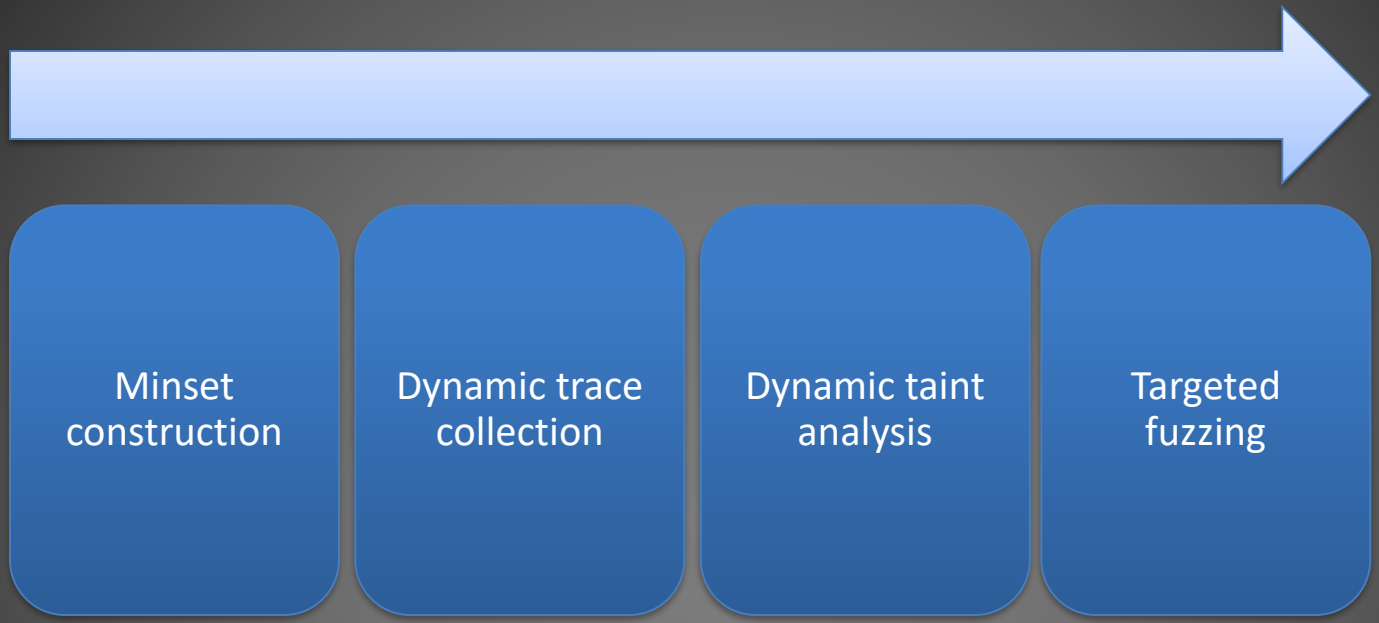
Objective

More vulnerabilities, more quickly

Microsoft

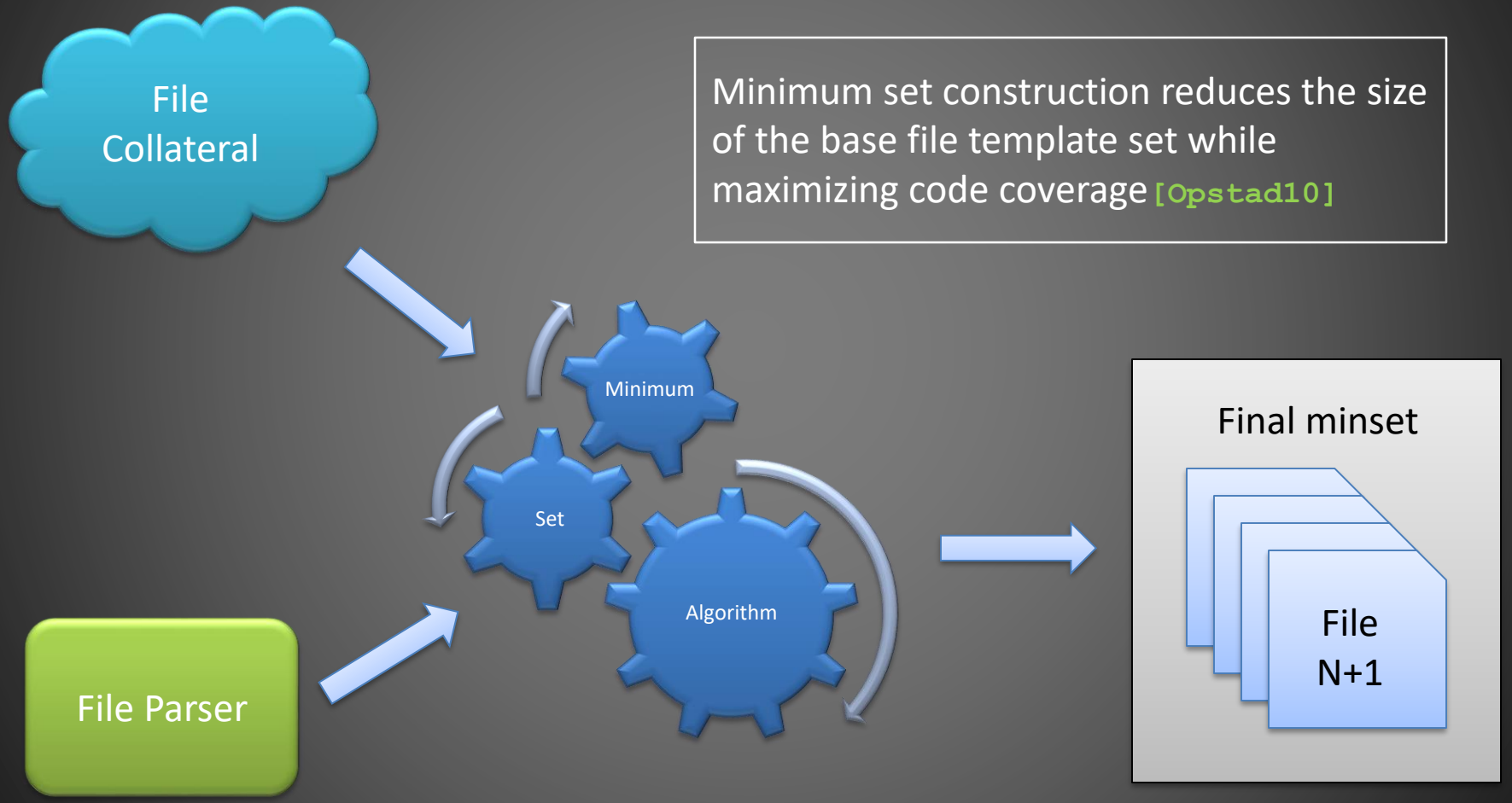
Key questions

- Is taint driven fuzzing effective?
- Does metric prioritization improve efficiency?
- Which metric & mutation strategy…
  - finds more distinct and unique vulnerabilities?
  - finds vulnerabilities most quickly?
  - finds higher severity vulnerabilities?
- Do crashes correspond to metrics?

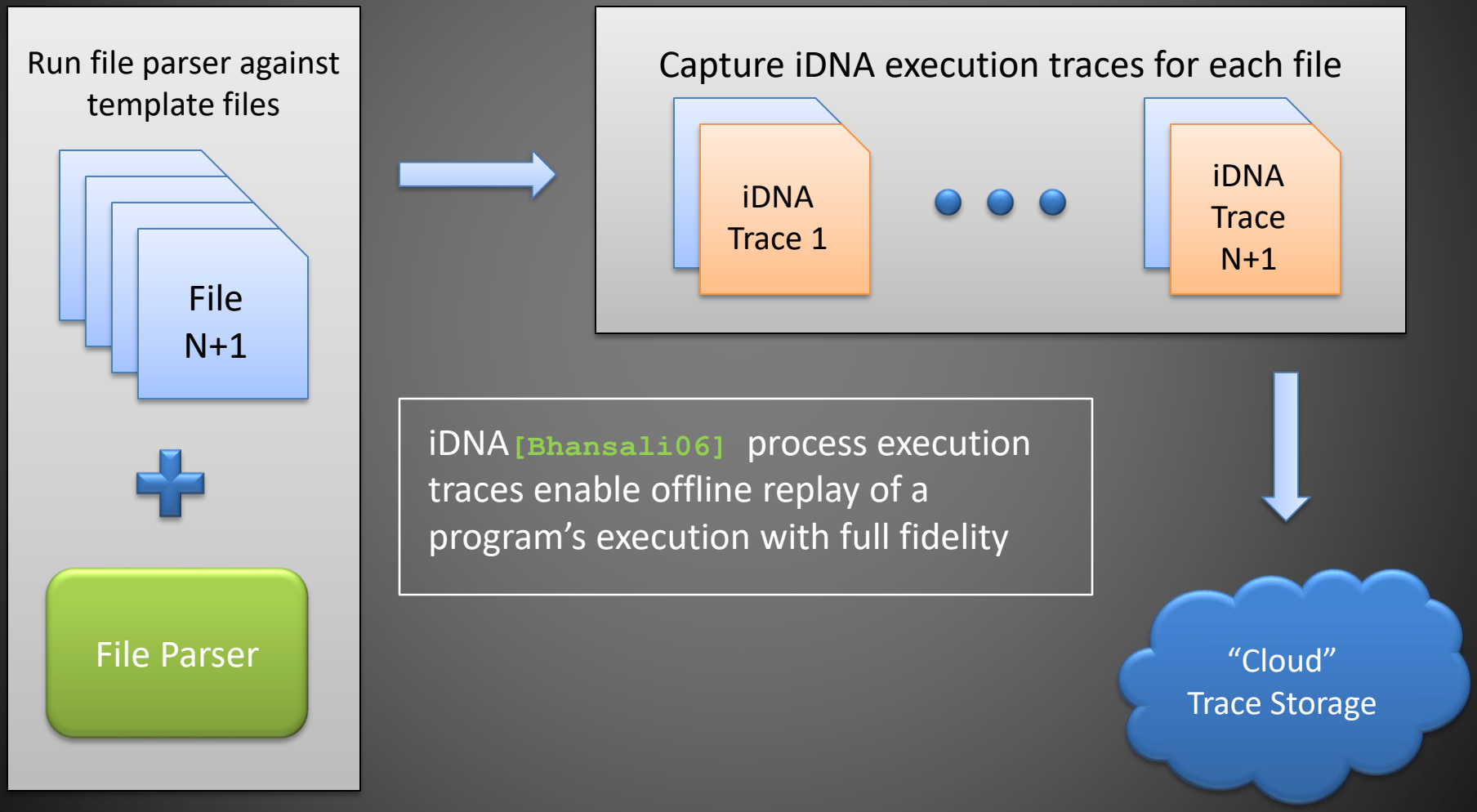


TARGETED TAINT DRIVEN FUZZING

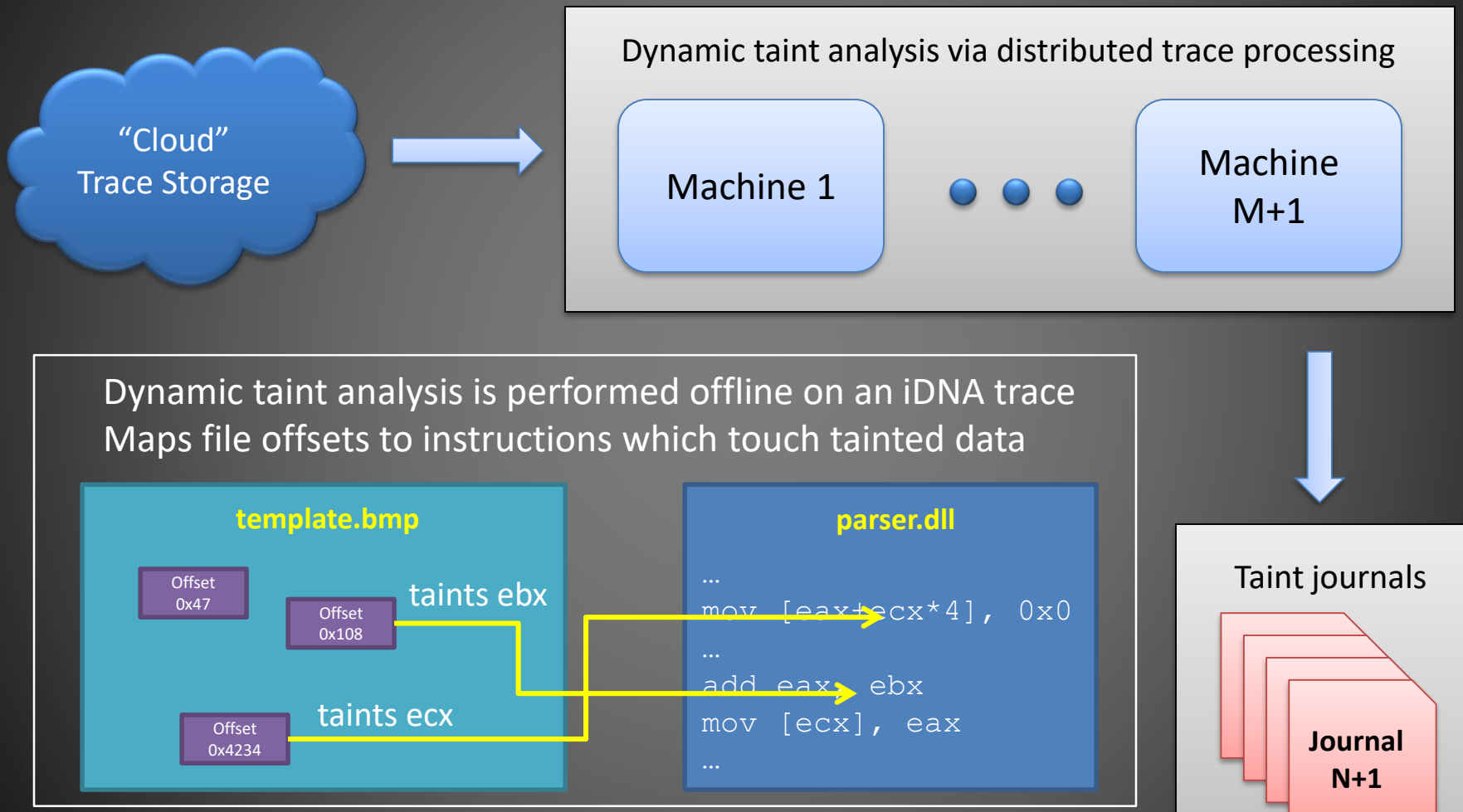
Minset construction



Dynamic trace collection



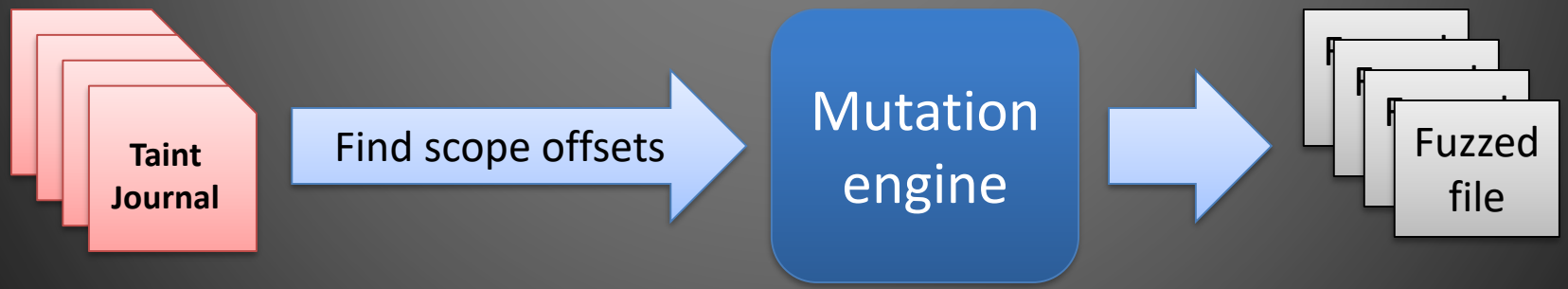
Dynamic taint analysis



Dynamic taint analysis [Schwartz10]

Taint driven fuzzing

- Fuzz offsets that taint a specific program scope
  - Binary = { foo.dll }
  - Functions = { foo, bar, … }
  - Instruction types = { “rep movsd”, “jcc” }
  - Source file = { parser.c }



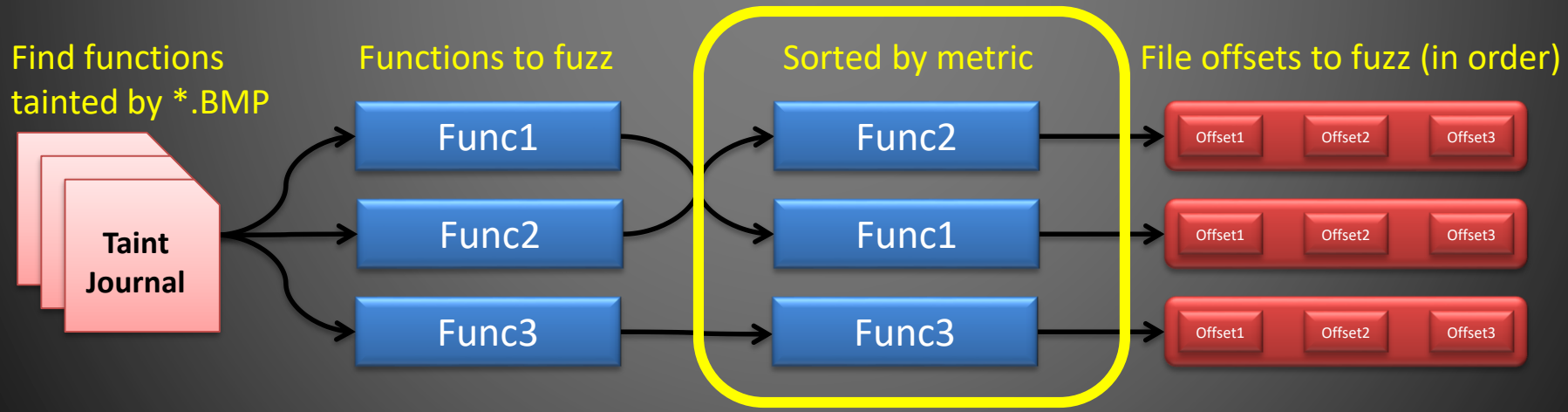
Related work: [Ganesh09] and [Iozzo10] also discuss directed fuzzing via taint data

SOFTWARE METRICS

Microsoft

Prioritizing offsets using software metrics

- Metrics can be used to sort program elements
  - Ex: Order functions by cyclomatic complexity
- Taint journals enables granular offset selection
  - Ex: Find offsets that taint functions foo, bar, …



Which software metrics can we use for sorting?

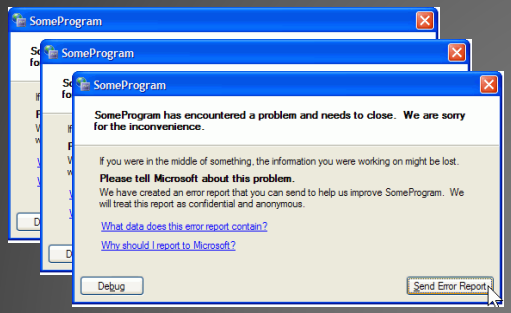
Cyclomatic complexity (CC)

$$M = E - N + 2P$$

E = # edges, N = # nodes, P = # connected components
M = cyclomatic complexity

- Well known software quality metric [McCabe76]
  - Measures independent paths through a flow graph
  - More paths = more complex
- Complexity metrics can predict defects [Nagappan05b]
- Targeted fuzzing via CC not a new idea [McCabe08, Iozzo10]
  - No empirical data has been provided, though

Crash reports



Windows Error Reporting

Crash reports indicate real world failures

(usually)

Hypothesis

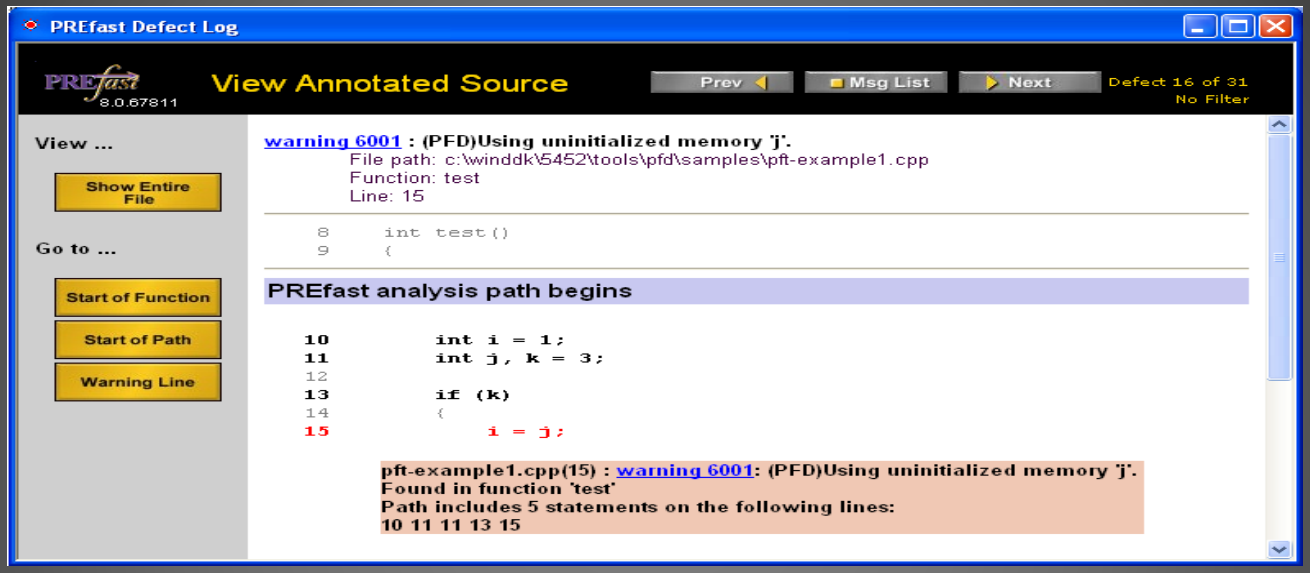
The more crash reports we see, the more likely it is that there is a reproducible defect

Observed crashes metric

Number of crashes that have been observed in a given program scope

Microsoft

Static analysis warnings



Hypothesis

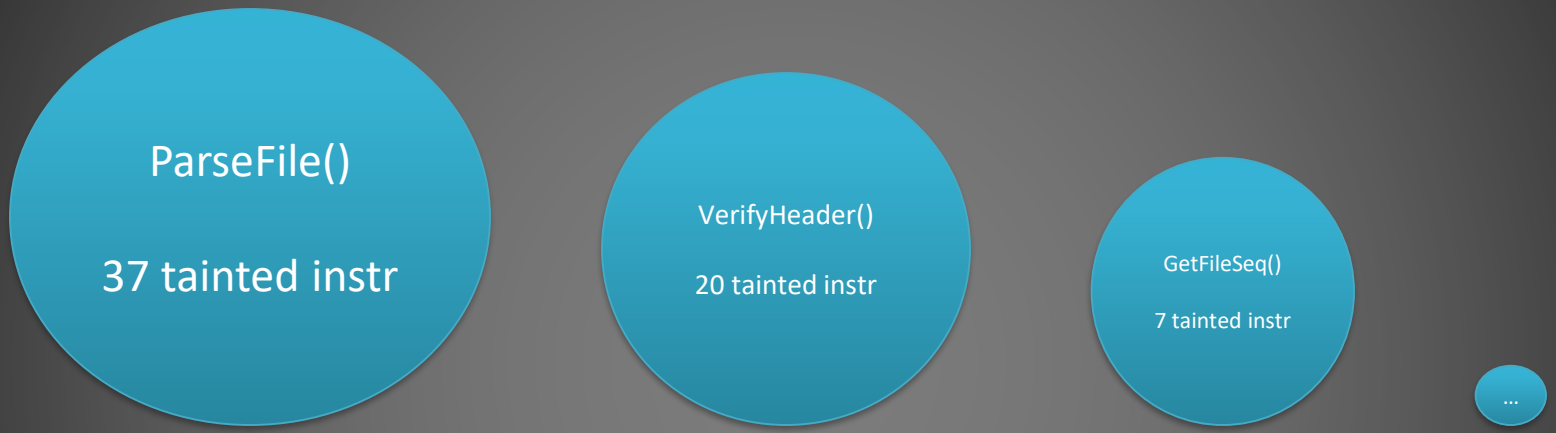
Static analysis warnings correlate with reproducible failures [Nagappan05]

Static analysis warning density metric

Number of static analysis warnings in a given program scope

Microsoft

Attack surface exposure



Hypothesis

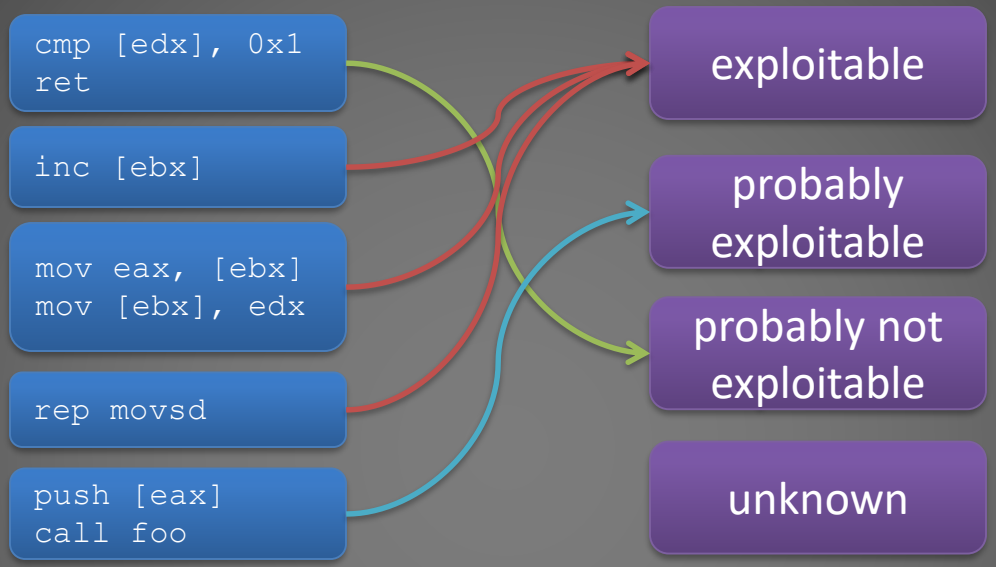
The more untrusted data a program deals with, the more likely it is that a defect will exist

Attack surface exposure metric

Number of instructions tainted by untrusted data in a given program scope

Exploitability

Simulate an exception at every instruction



Measure worst case exploitability

Hypothesis

Program scopes with a higher density of exploitable instruction sequences are more likely to have exploitable (high risk) vulnerabilities

Exploitability metric

Average worst case exploitability of instructions in a given program scope

EXPERIMENTAL RESULTS & ANALYSIS

Microsoft

Experiment setup

4 binary file format parsers	A (~33,000 tainted instructions) B (~10,000 tainted instructions) C (~23,000 tainted instructions) D (~217,000 tainted instructions)
5 fuzzing engines	3 taint driven engines 2 control engines
6 metrics Program scope: tainted functions in parser binary	• Cyclomatic complexity • Observed crashes • Static analysis warning density • Attack surface exposure • Exploitability • No metric (control)
5 days of fuzzing (maximum)	Upper bound, may finish earlier
77* total runs	1 run = engine + metric + target
Distinct crashes	Classified by major hash [Shirk08]
Unique crashes	Distinct crashes found only by a specific fuzzer

* No static analysis data for target D

Microsoft

Fuzzing engines & mutation strategies

Fuzzer engine	Type	Mutation strategy
Single Byte	Taint Driven Mutation	Mutates a single tainted byte at a time using a fixed set of fuzz values
Cerberus	Taint Driven Mutation	A three pronged approach: 1. Single Byte fuzzing for offsets with less than 4 contiguous tainted bytes 2. DWORD fuzzing for offsets with 4 contiguous tainted bytes 3. Random substitution of a random number of bytes within a tainted sub-region for offsets with more than 4 contiguous tainted bytes
Cerberus Lite	Taint Driven Mutation	Only approach #3 from Cerberus (no DWORD or Single Byte)
Charlie*	Mutation	Mutates a random number of offsets at a time using random substitution [Miller10]
FileFuzzer 3*	Mutation	Mutates a file using multiple byte random substitution (possibly growing the file) [Shirk08]

* Control fuzzer for this experiment

Distinct crashes per run

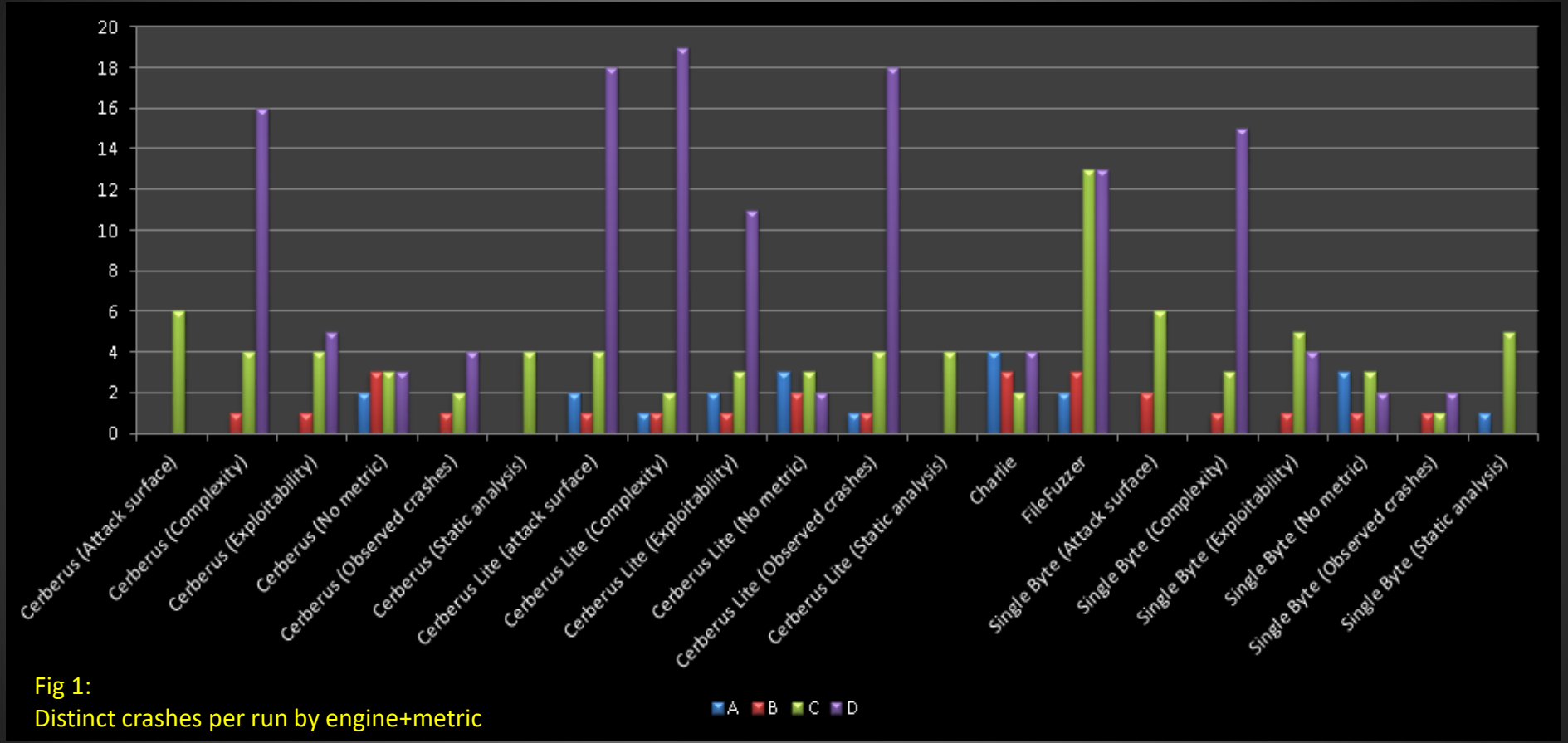


Fig 1:
Distinct crashes per run by engine+metric

Winners by target

A	B	C	D
Charlie (4)	Tie (3); Charlie, Cerberus, FileFuzzer	FileFuzzer (13)	Cerberus Lite + Complexity (19)

How effective is taint driven fuzzing?

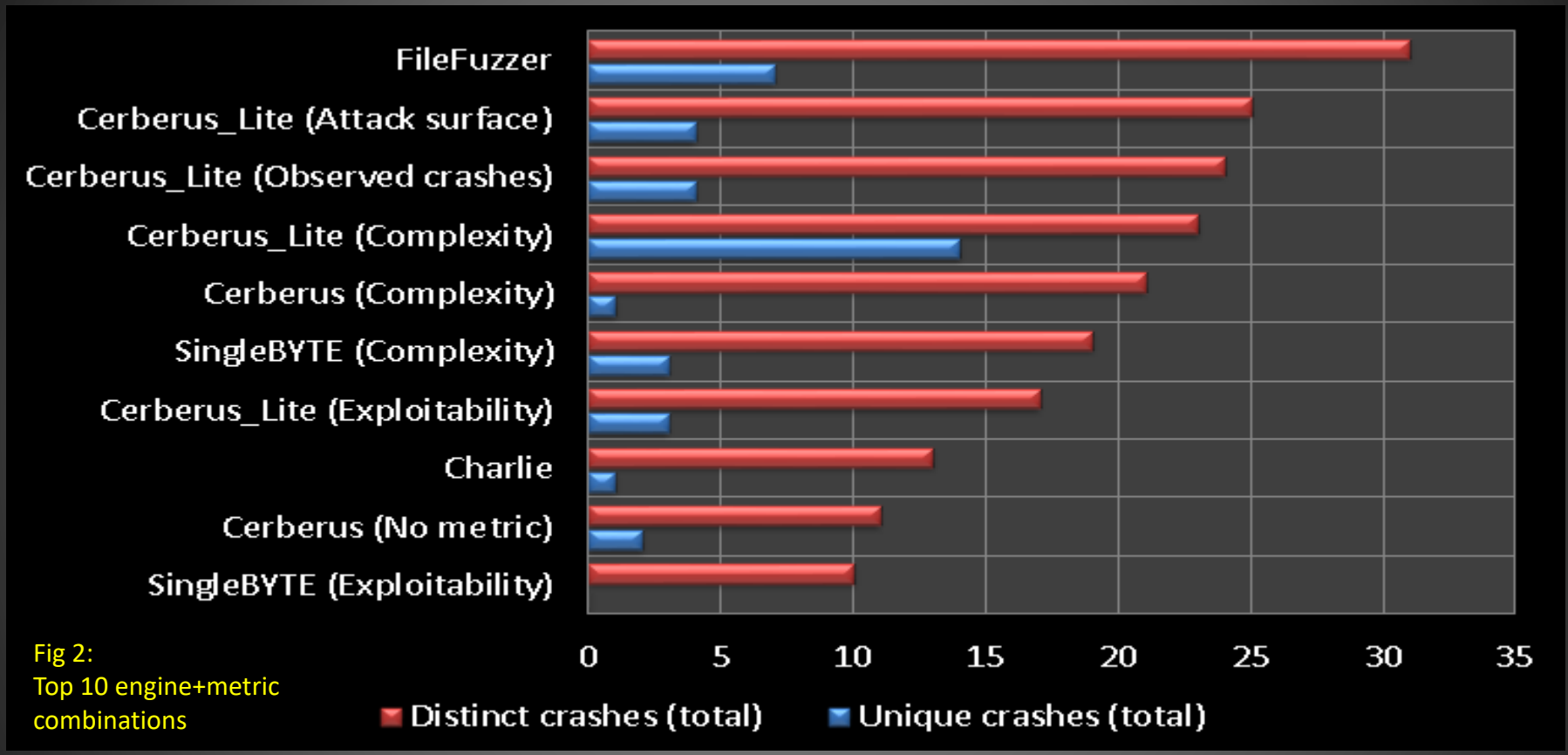


Fig 2:
Top 10 engine+metric combinations

Observations:

- FileFuzzer found more distinct crashes, but Cerberus Lite + Complexity found more unique
- Prioritizing by cyclomatic complexity consistently beat other metrics regardless of engine
- Taint and control engine effectiveness varied by target (breakdown included in appendix)

Does metric prioritization help?

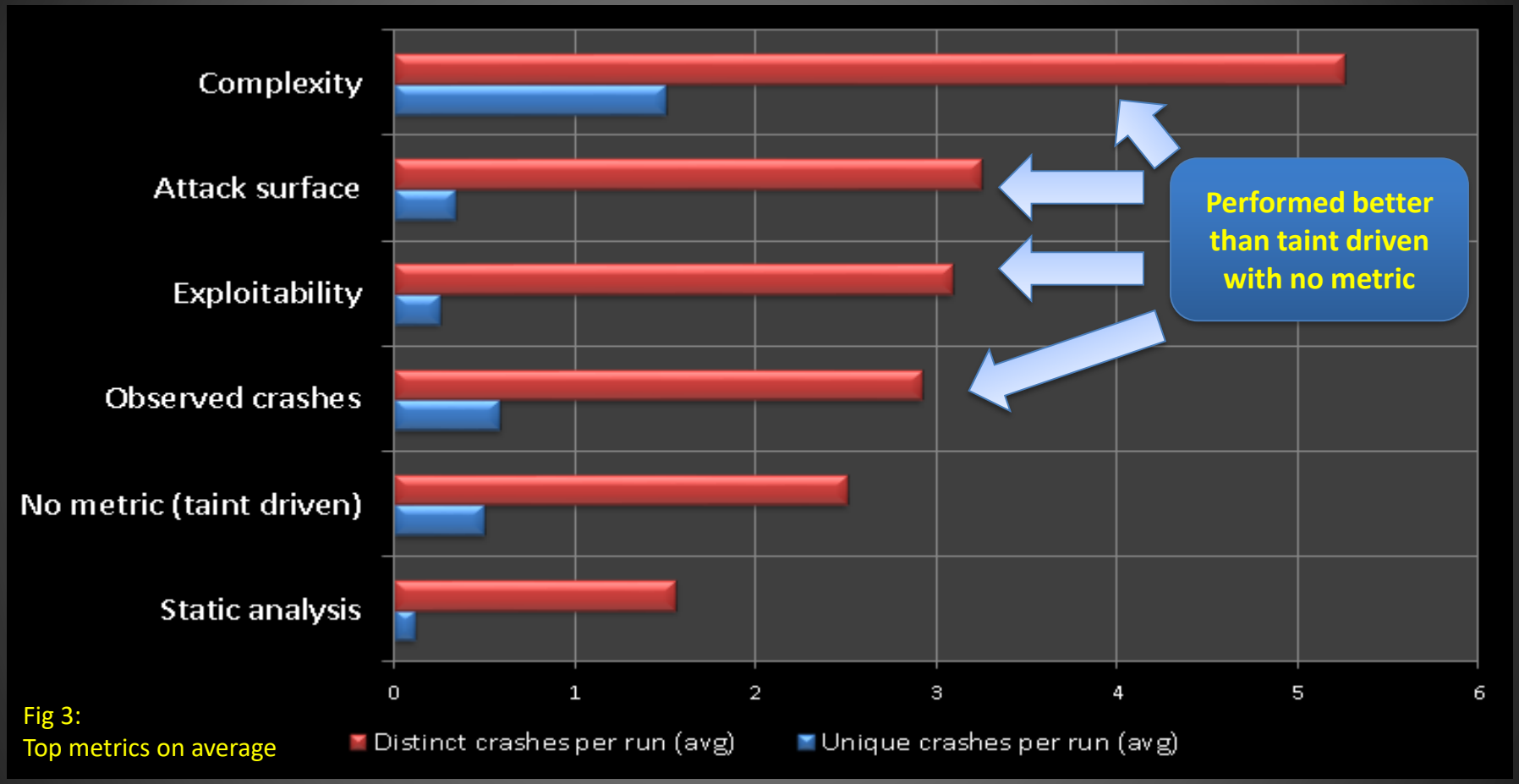


Fig 3:
Top metrics on average

Observations:

- Metric prioritization performed better on average than no prioritization for most metrics
- Static analysis was the only metric that performed worse than no metric (on average)

Metric prioritization doesn't always help

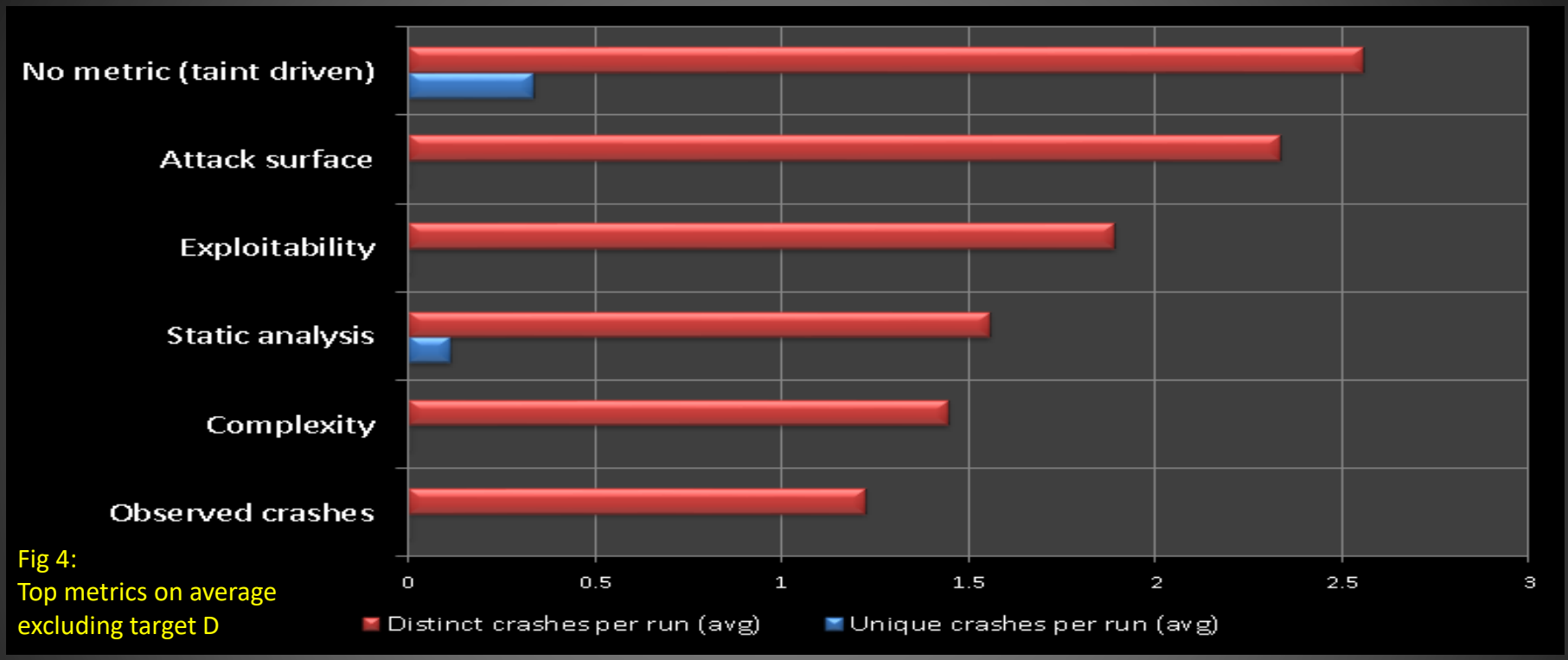


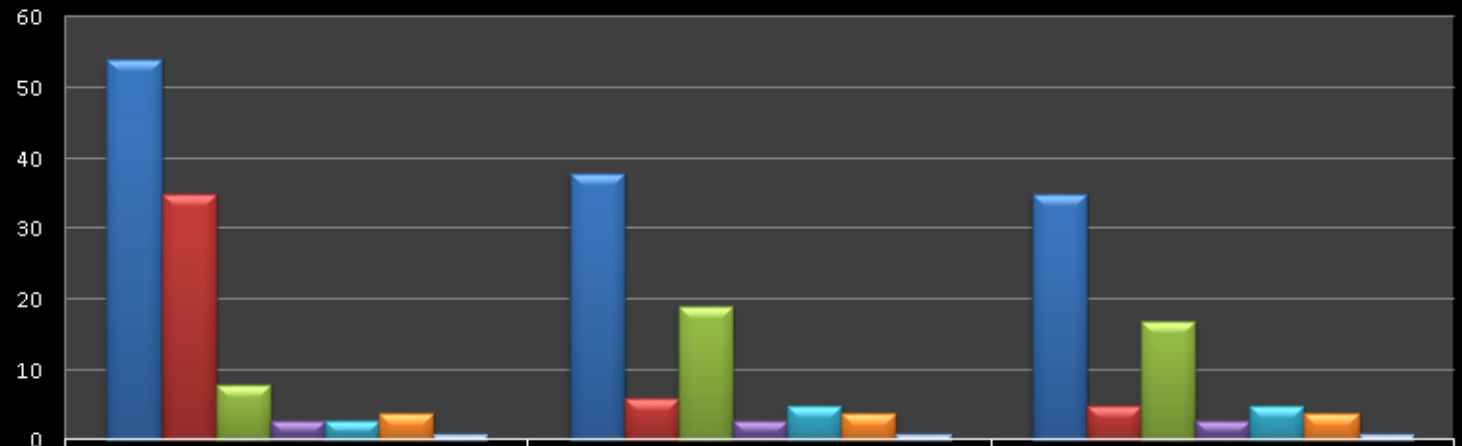
Fig 4:
Top metrics on average excluding target D

Observations:

- We found that target D heavily dominated our findings
- Excluding target D showed that all metrics performed *worse* than no metric on average
- Our analysis suggests this is because
  - Most metrics had a shorter running time than "no metric" (correlated with crashes found)
  - Targets A/B/C are much smaller parsers than target D (prioritization is thus less impactful)

Crash overlap drill down by engine

Fig 5:
Crash overlap by engine



	Cerberus_Lite	Cerberus	SingleBYTE
Total distinct crashes	54	38	35
Found only by this engine	35	6	5
Found by 1 other engine	8	19	17
Found by 2 other engines	3	3	3
Found by 3 other engines	3	5	5
Found by 4 other engines	4	4	4
Found by 5 other engines	1	1	1

Observations:

- Cerberus Lite was the best performing taint driven engine on average (35 unique crashes)
- Out of 103 distinct crashes, taint driven engines were the only ones to find 66 of them
- Control fuzzers are excluded from this comparison for fairness reasons
  - Taint driven engines had 6x the opportunity (for each metric)

Do metrics find issues sooner?

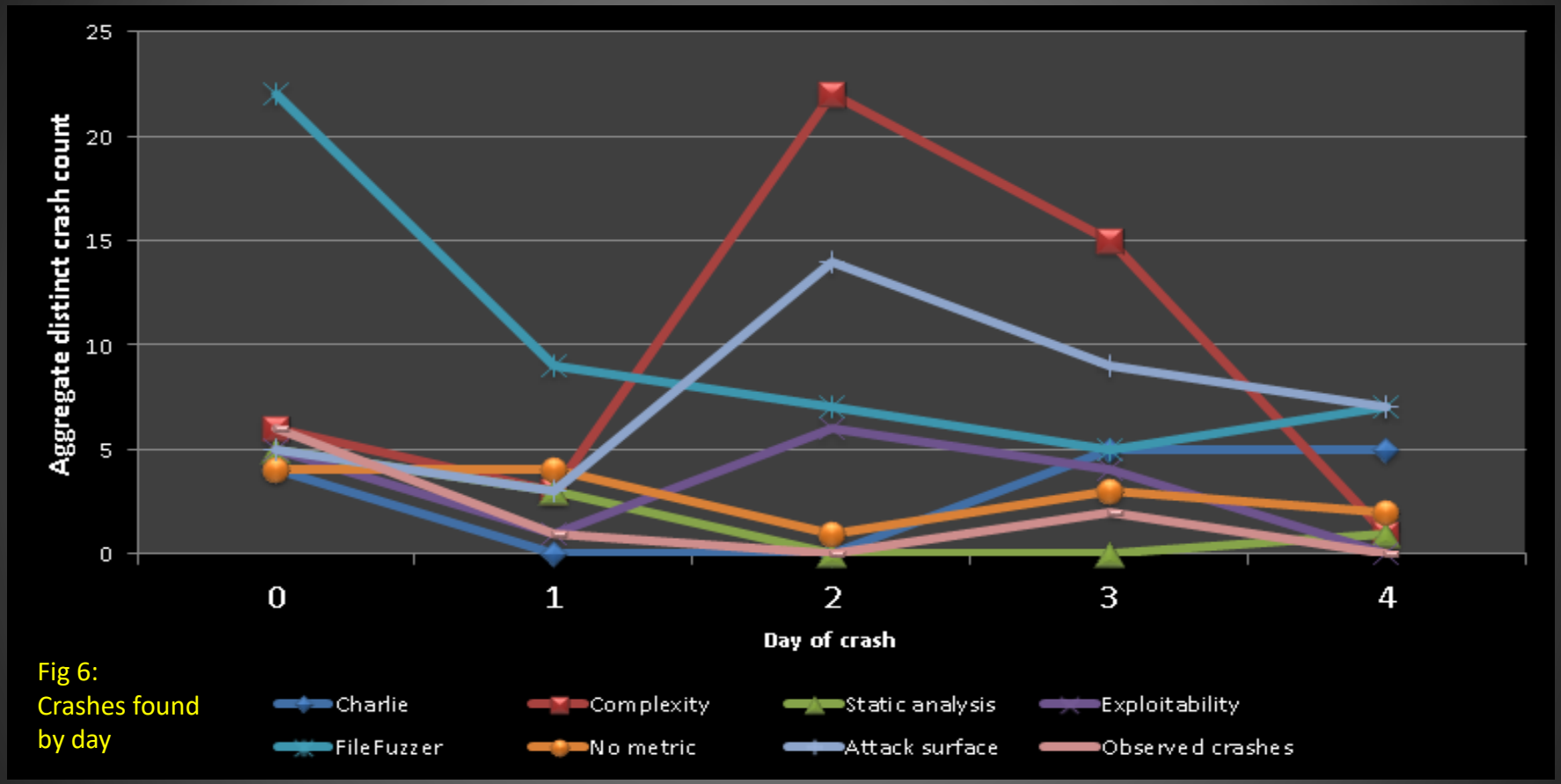


Fig 6:
Crashes found by day

Observations:

- FileFuzzer found most of its distinct crashes on the first day of fuzzing
- Upfront sorting costs delayed metric findings (as much as 48 hours in some cases)

Do metrics find higher severity issues?

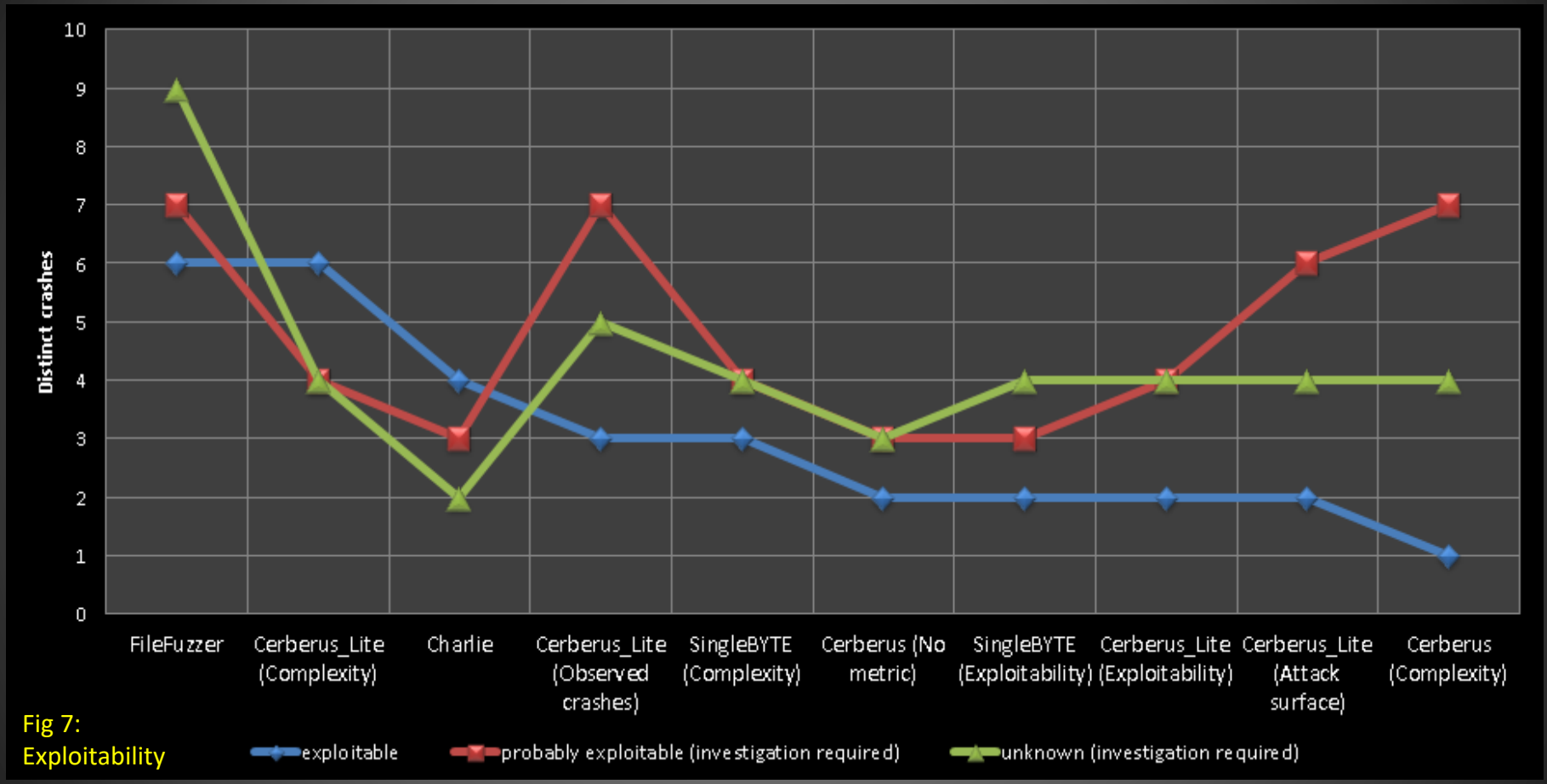


Fig 7: Exploitability

Observations:

- No indication from our dataset that metrics find higher severity issues

How “targeted” is taint driven fuzzing?

85 distinct crashes found by taint driven engines

7 of 85

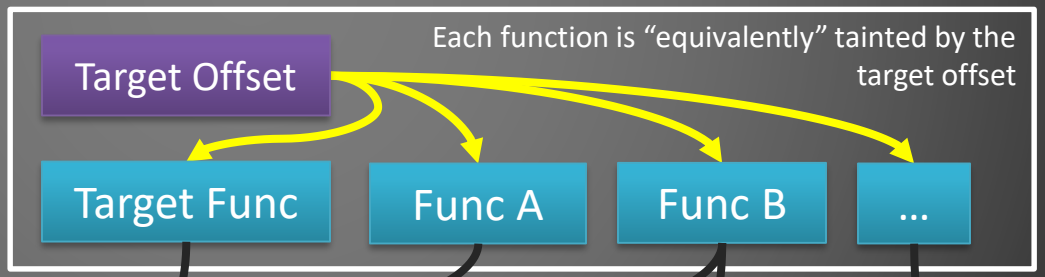
Crash Func == Targeted Func

- 4 found by attack surface metric
- 4 found by observed crashes metric
- 2 found by no metric (taint driven)
- 1 found by cyclomatic complexity metric

Some overlap between metrics

13 of 85

Crash Func =~ Targeted Func



24 of 85

Crash Func =~ Immediate callee (of targeted func)

Microsoft

Do crashes correspond to metrics?

Static analysis warning density	• 6 distinct crashes
	• **2 out of 6 confirmed the static analysis warning**
	  • 1 integer wrap leading to trunc alloc warning
	  • 1 unbounded write warning

Observed crashes	• 28 distinct crashes • 4 out of 28 crashed in the targeted function • **1 out of 4 confirmed the observed crash**

Software defect warnings can be reproduced and confirmed through targeted taint driven fuzzing

Summary of findings

Most distinct crashes	• FileFuzzer (31)
Most unique crashes	• Cerberus Lite + Cyclomatic complexity (14)
Best overall control engine	• FileFuzzer (31 distinct, 7 unique)
Best overall taint driven engine + metric	• Cerberus Lite + Attack surface (25 distinct, 4 unique)
Best overall taint driven engine	• Cerberus Lite (54 distinct, 35 unique)

CONCLUSIONS

Limitations & future work

- Limitations
  - Small sample size (only 4 targets)
  - Short run time (only 5 days)
- Future work
  - Expand sample size
  - Experiment with additional metrics
  - Gap analysis on crashes found only by control engines
  - Optimization of sorting procedures for metrics

Conclusion

- Taint driven fuzzing has numerous benefits
  - Granular targeting capabilities
  - Insight into what was covered (and not covered) during fuzzing
- Our research indicates that
  - Taint driven fuzzing is an effective fuzzing technique
  - Metrics can improve effectiveness, but not for all targets
  - Larger & more complex targets benefit more from metrics
- Fuzzer diversification is still important
  - Performance of fuzzers differs based on the target
  - Control fuzzers found issues taint driven did not (and vice versa)

Acknowledgements

- Nitin Kumar Goel & Mark Wodrich (MSRC)
  - Dynamic taint analysis tool design & development
- Gavin Thomas (MSRC)
  - Targeted taint driven fuzzing architect
- Peter Beck, Dave Weinstein, Jason Shirk
  - Crash exploitability analysis tools & research
- Nachi Nagappan (MSR) & Thirumalesh Bhat
  - Software metric tools & research
- Saurabh Boyed
  - Distributed trace analysis infrastructure

Questions?

Did you think this talk was cool?
Want to work with brilliant people on fascinating security projects?
http://careers.microsoft.com

Learn more about Security Science at Microsoft
http://www.microsoft.com/security/msec

Security Research & Defense blog
http://blogs.technet.com/srd

Microsoft

References

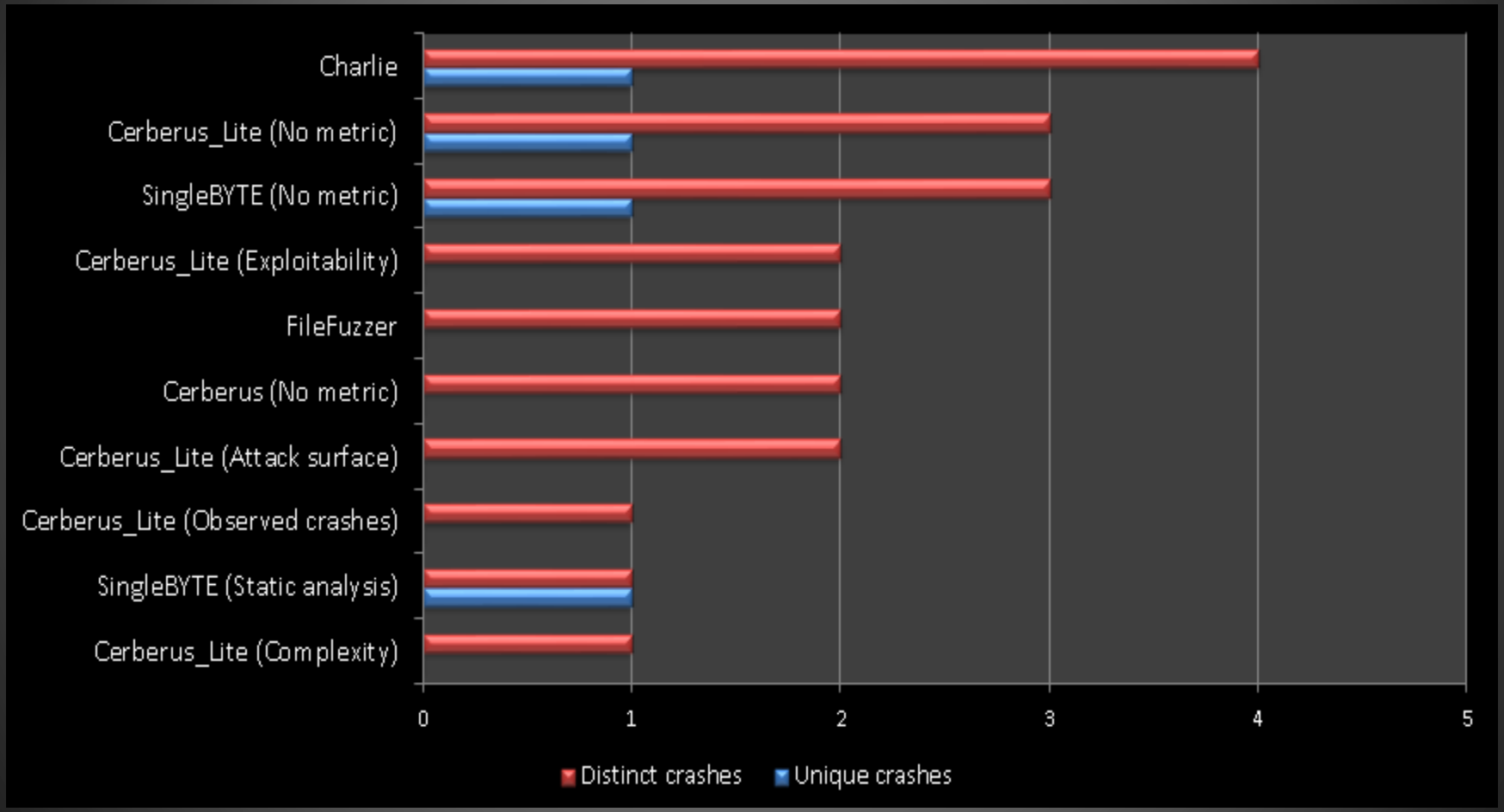
[McCabe76]	Thomas McCabe. A complexity measure. IEEE TSE, 1976.
[Nagappan05]	Nachiappan Nagappan, Thomas Ball. Static analysis tools as early indicators of pre-release defect density. ICSE 2005.
[Nagappan05b]	Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. Microsoft Technical Report, 2005.
[Bhansali06]	Sanjay Bhansali et al. Framework for instruction-level tracing and analysis of program executions. VEE 2006.
[Drewry07]	Will Drewry and Tavis Ormandy. Flayer: Exposing Application Internals. WOOT 2007.
[Godefroid08]	Patrice Godefroid, Michael Levin, and David Molnar. Automated Whitebox Fuzz Testing. NDSS 2008.
[Miller08]	Charlie Miller. Fuzz by Number. CanSecWest 2008.
[McCabe08]	McCabe Software, Inc. Combining McCabe IQ with Fuzz Testing. 2008.
[Shirk08]	Jason Shirk, Lars Opstad, and Dave Weinstein. Fuzzed enough? When it's OK to put the shears down. Blue Hat 2008 (fall session).
[Eddington09]	Michael Eddington. Demystifying Fuzzers. Black Hat USA 2009.
[Ganesh09]	Vijay Ganesh, Tim Leek, Martin Rinard. Taint-based Directed Whitebox Fuzzing. ICSE 2009.
[Schwartz10]	Edward Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. Oakland 2010.
[Opstad10]	Lars Opstad. Using code coverage to improve fuzzing results. SRD Blog 2010.
[Iozzo10]	Vincenzo Iozzo. 0-Knowledge Fuzzing. Black Hat DC 2010.
[Molnar10]	David Molnar, Lars Opstad. Effective fuzzing strategies. 2010.
[Miller10]	Charlie Miller. Babysitting an army of monkies: fuzzing 4 products with 5 lines of python. CanSecWest 2010.
[Miller10b]	Charlie Miller et al. Crash analysis with BitBlaze. Black Hat USA 2010.
[WER]	Microsoft Windows Error Reporting: About WER

Additional data

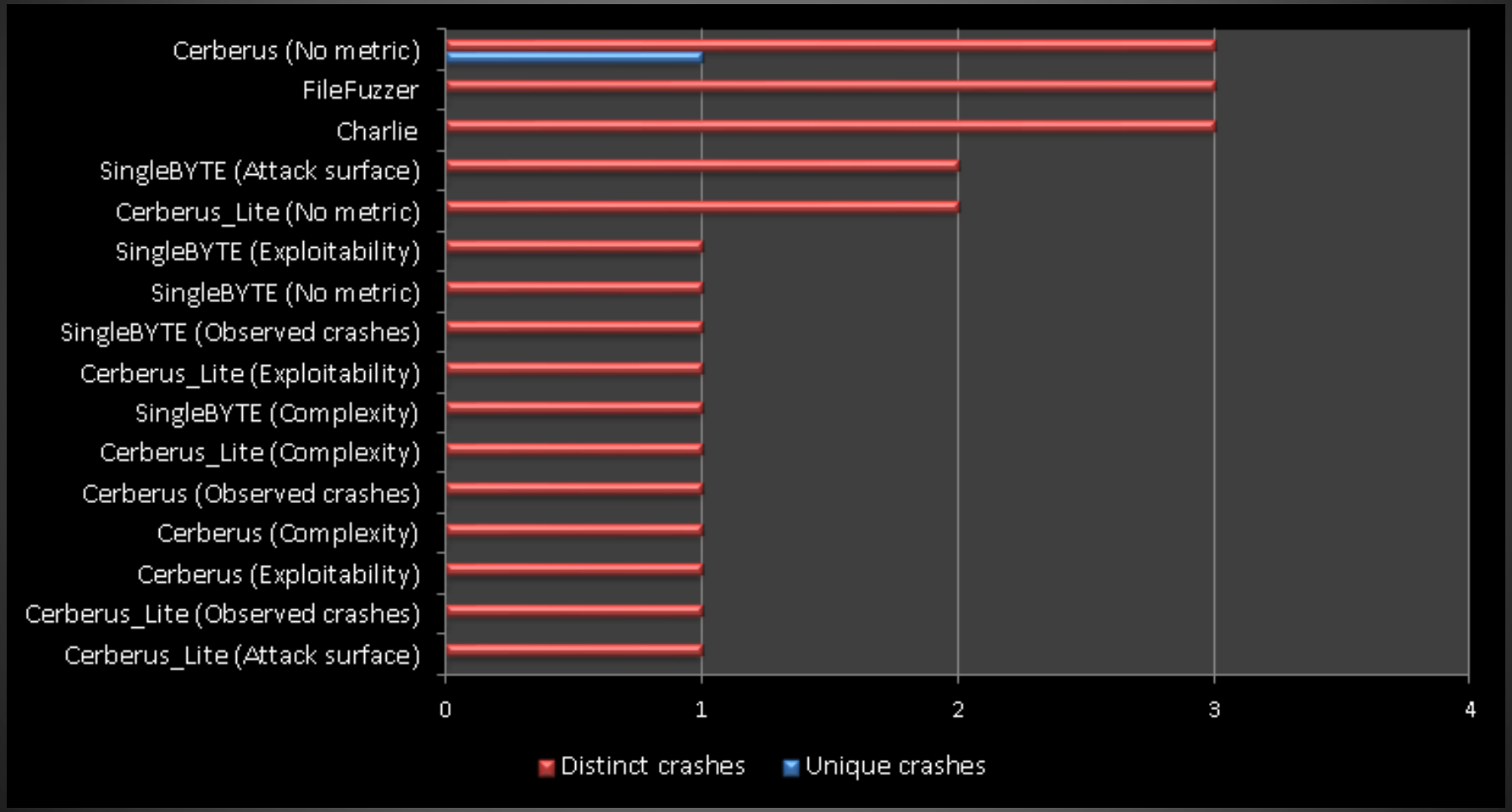
APPENDIX

Microsoft

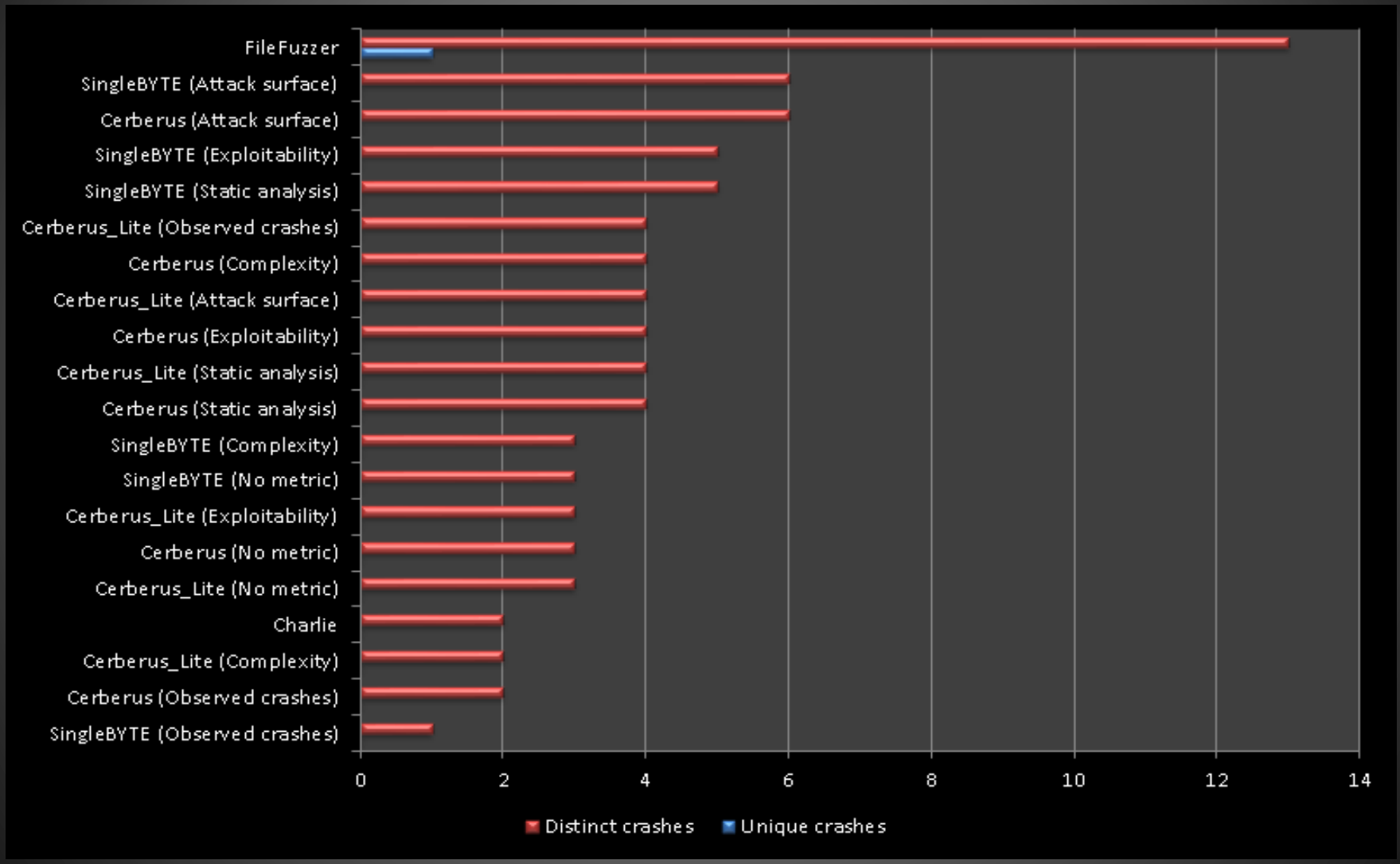
Target A crash breakdown



Target B crash breakdown



Target C crash breakdown



Microsoft

Target D crash breakdown