RUB

CASA
SECURING THE DIGITAL SOCIETY

Chair for
Software Security

Who the Fuzz Cares About Code Coverage?

Kevin Borgolte

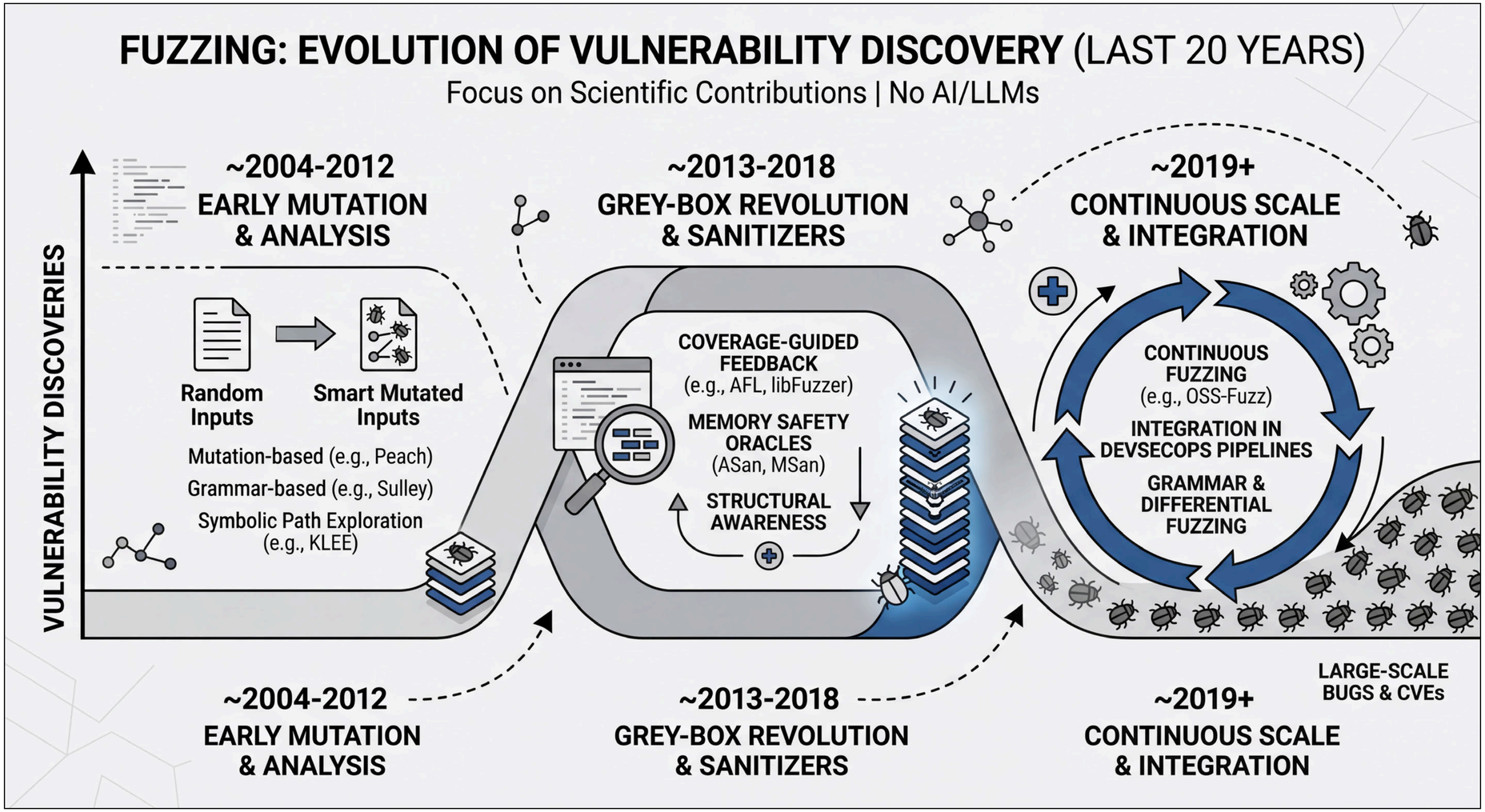
kevin.borgolte@rub.de

April 12, 2026
Search-Based and Fuzz Testing

Who the Fuzz Is This Person?

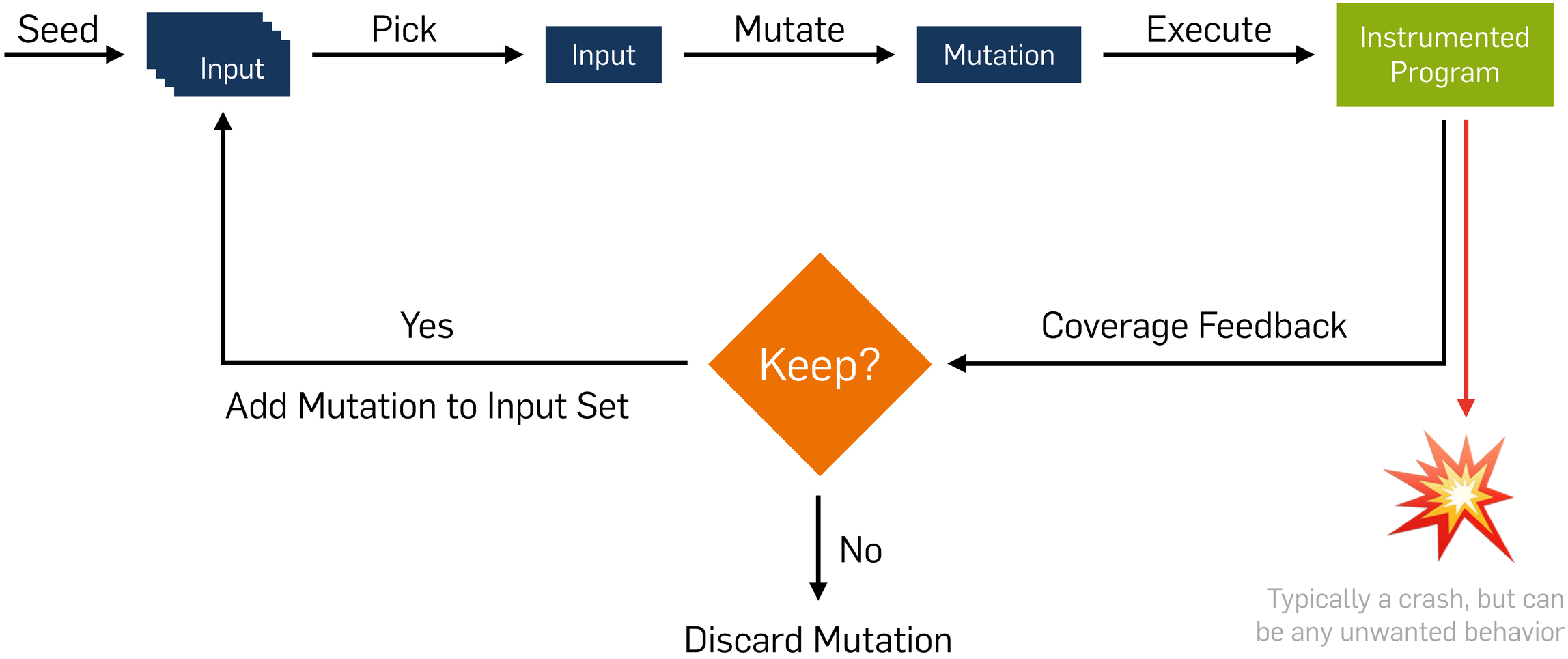
- Professor at Ruhr University Bochum since 2021; before TU Delft, Princeton, and UC Santa Barbara
  - Chair for Software Security (softsec.rub.de) and part of the DFG Excellence Cluster CASA (casa.rub.de)
- Real-world (networked) software security
  - Vulnerability discovery/exploitation/defenses
  - Network protocol implementations and interactions
  - Some research/vulnerabilities are public, some are not :)
- Forever ago
  - DARPA Cyber Grand Challenge (CGC) with Shellphish (2016; not AIxCC)

Fuzzing Since 2004 (As Seen by Nano Banana Pro 2)

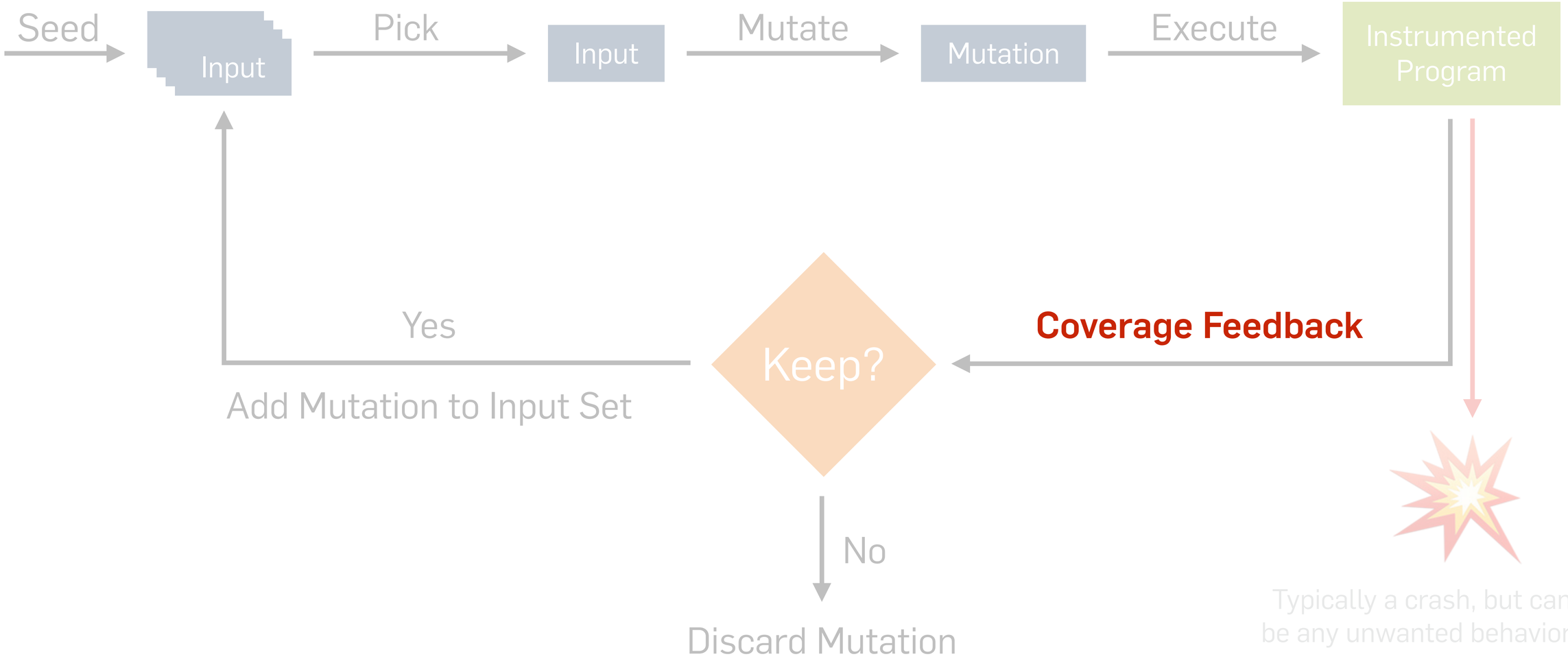


Prompt: I need a visualization that illustrates how fuzzing has increased and improved vulnerability discovery over the last 10-20 years. Make sure that this is simple in design, not busy, has somewhat little text and it is not overly colorful. It also MUST not include AI/LLMs and focus on the scientific contributions.

Coverage-guided Fuzzing



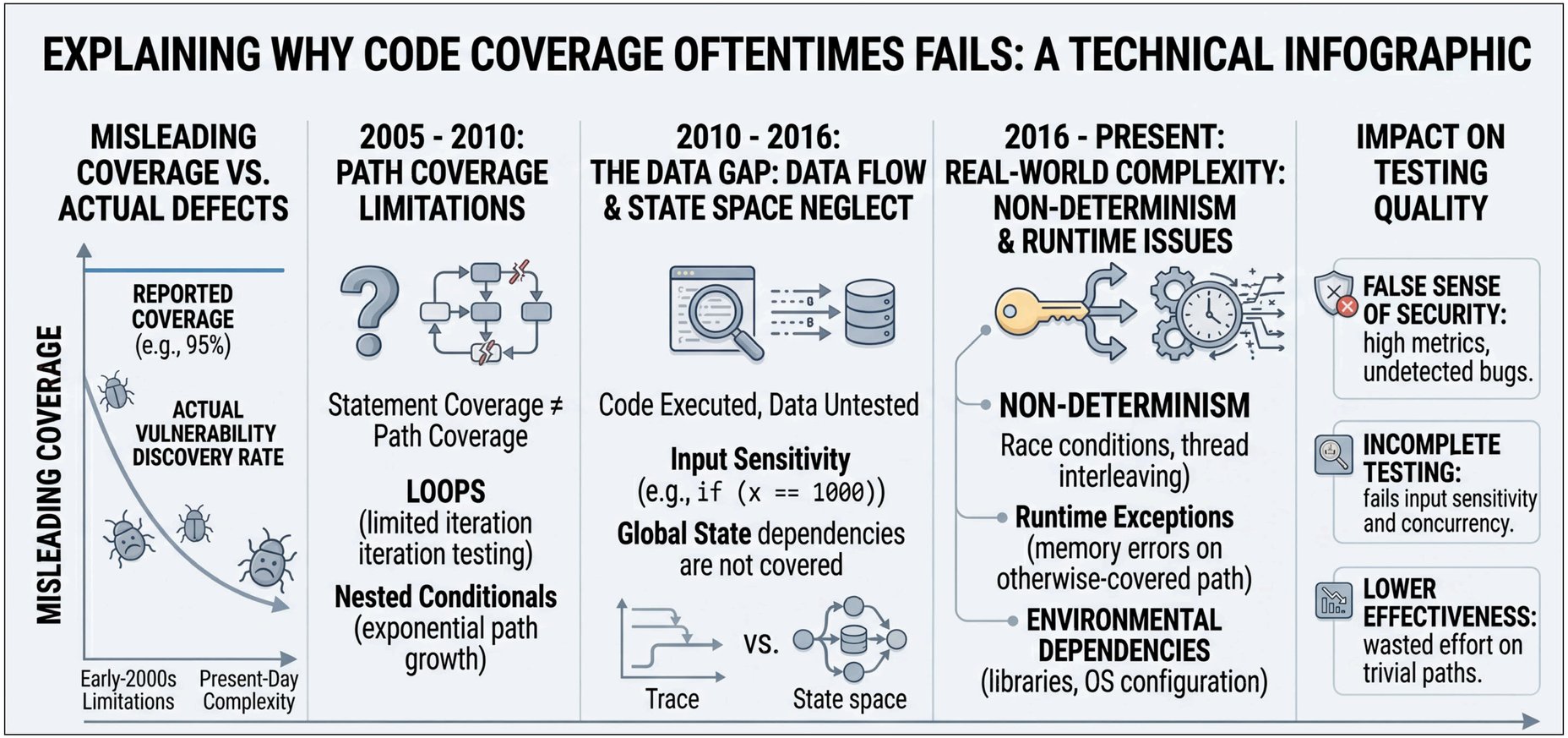
Coverage-guided Fuzzing



Coverage-guided Fuzzing and Coverage Feedback

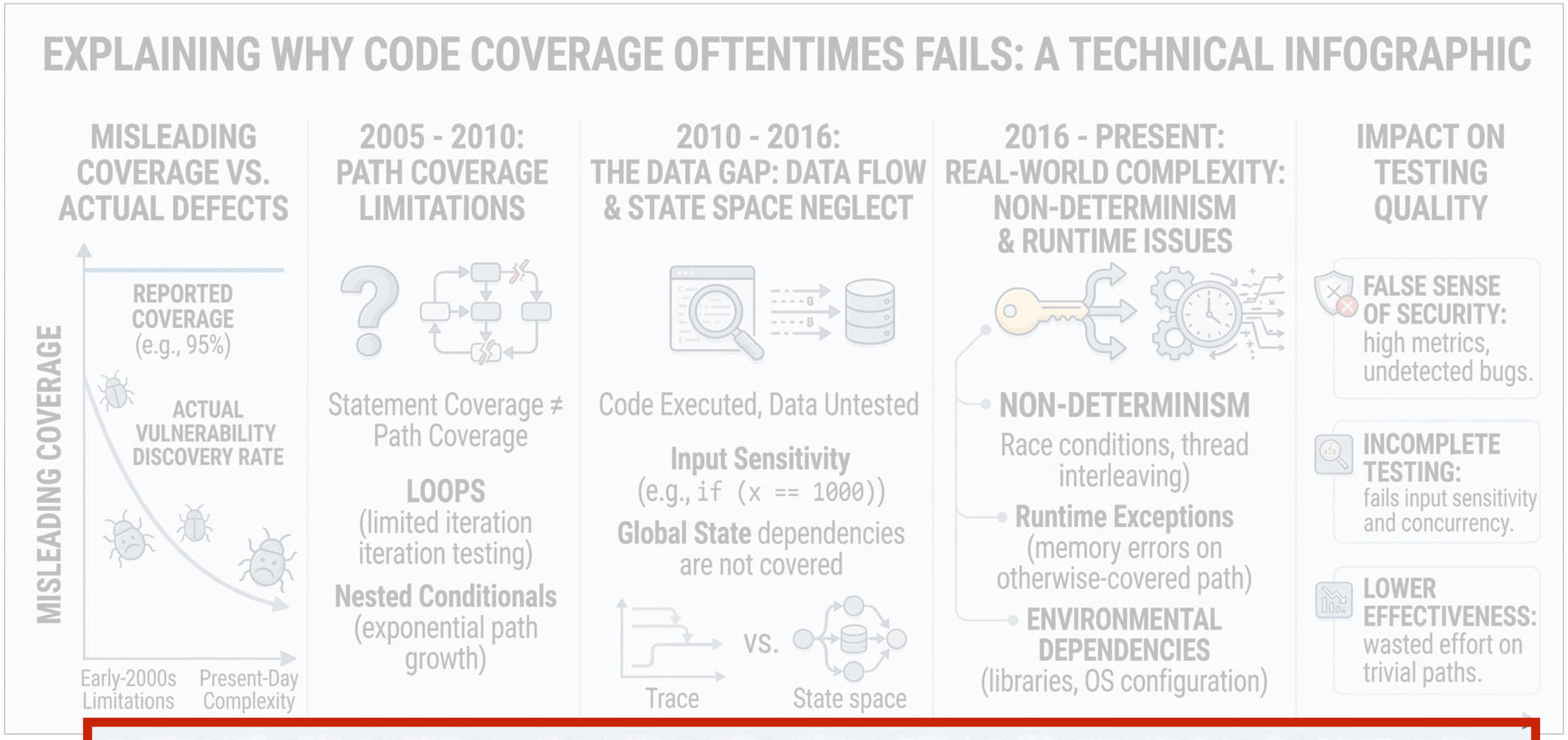
- Coverage feedback is almost always code coverage
  - Edge coverage vs. branch coverage vs. path coverage (all the same)
  - This is what we will expand on a bit today
- To make it a bit more visual: You can somewhat imagine it as tracking through which doors in a building we went (or which path we took through a building).

Code Coverage Ain't Enough (As Seen by Nano Banana Pro 2)



Prompt: In the same style, I need an explanation of why code coverage oftentimes fails.

Code Coverage Ain't Enough (As Seen by Nano Banana Pro 2)



Prompt: In the same style, I need an explanation of why code coverage oftentimes fails.

I love that Nano Banana Pro 2 kept the initial "No AI/LLMs" here from the initial prompt, but they actually can be useful for comprehensive exploration.

Useful Heuristics?

- Even AI agrees:
  Code coverage is only a useful heuristic, not a bug detector.
- Yet, our benchmarking for fuzzers focuses almost exclusively on improving code coverage.
- Sometimes, we investigate at bug-based benchmarks, but there are other problems with that, which we talk about later.

By avg. score		By avg. rank	
	average normalized score		average rank
fuzzer		fuzzer	
libafl	98.01	aflplusplus	2.61
honggfuzz	94.33	libafl	3.87
libfuzzer	94.26	libfuzzer	4.48
aflplusplus	92.22	honggfuzz	5.13
afl	85.42	afl	5.70

Goodhart's Law

"Any observed statistical regularity will tend to collapse once pressure is placed upon it for control purposes."

"Problems of Monetary Management: The UK Experience", C. Goodhart

Goodhart's Law

Covering more code means we find more (interesting) bugs

*"Any observed **statistical regularity** will tend to collapse once pressure is placed upon it for control purposes."*

"Problems of Monetary Management: The UK Experience", C. Goodhart

Goodhart's Law

Covering more code means we find more (interesting) bugs

*"Any observed **statistical regularity** will **tend to collapse once pressure is placed upon it for control purposes**."*

"Problems of Monetary Management: The UK Experience", C. Goodhart

Us maximizing code coverage for finding bugs/vulnerabilities?

Goodhart's Law

*"**When a measure becomes a target, it ceases to be a good measure.** The more [...] performance becomes an expectation, the poorer it becomes as a discriminator of individual performances. [...] targets that seem measurable become enticing tools for improvement. [...] **It was that conflation of 'is' and 'ought',** alongside quantifiable written assignments, which led in Hoskin's view to [...] accountability. **This was articulated [...] for the first time around 1800 as 'the awful idea of accountability'.**"*

"'Improving ratings': audit in the British University system", M. Strathern

"'Improving ratings': audit in the British University system", M. Strathern, European Review, Volume 5, Issue 3, July 1997
DOI: 10.1002/(SICI)1234-981X(199707)5:3<305::AID-EURO184>3.0.CO;2-4

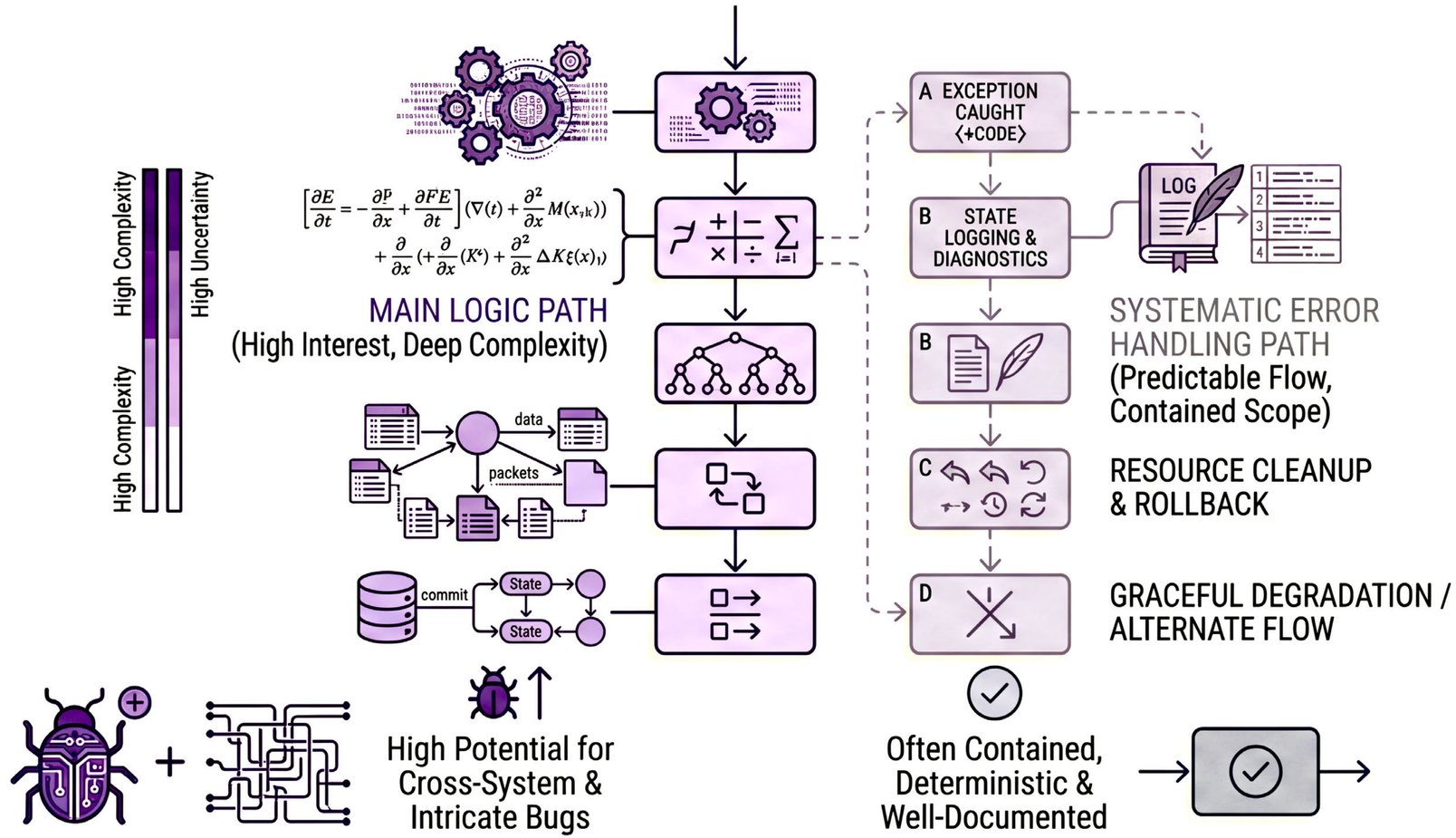
Comprehensive Testing?

Should our goal not be to

find as many bugs as possible

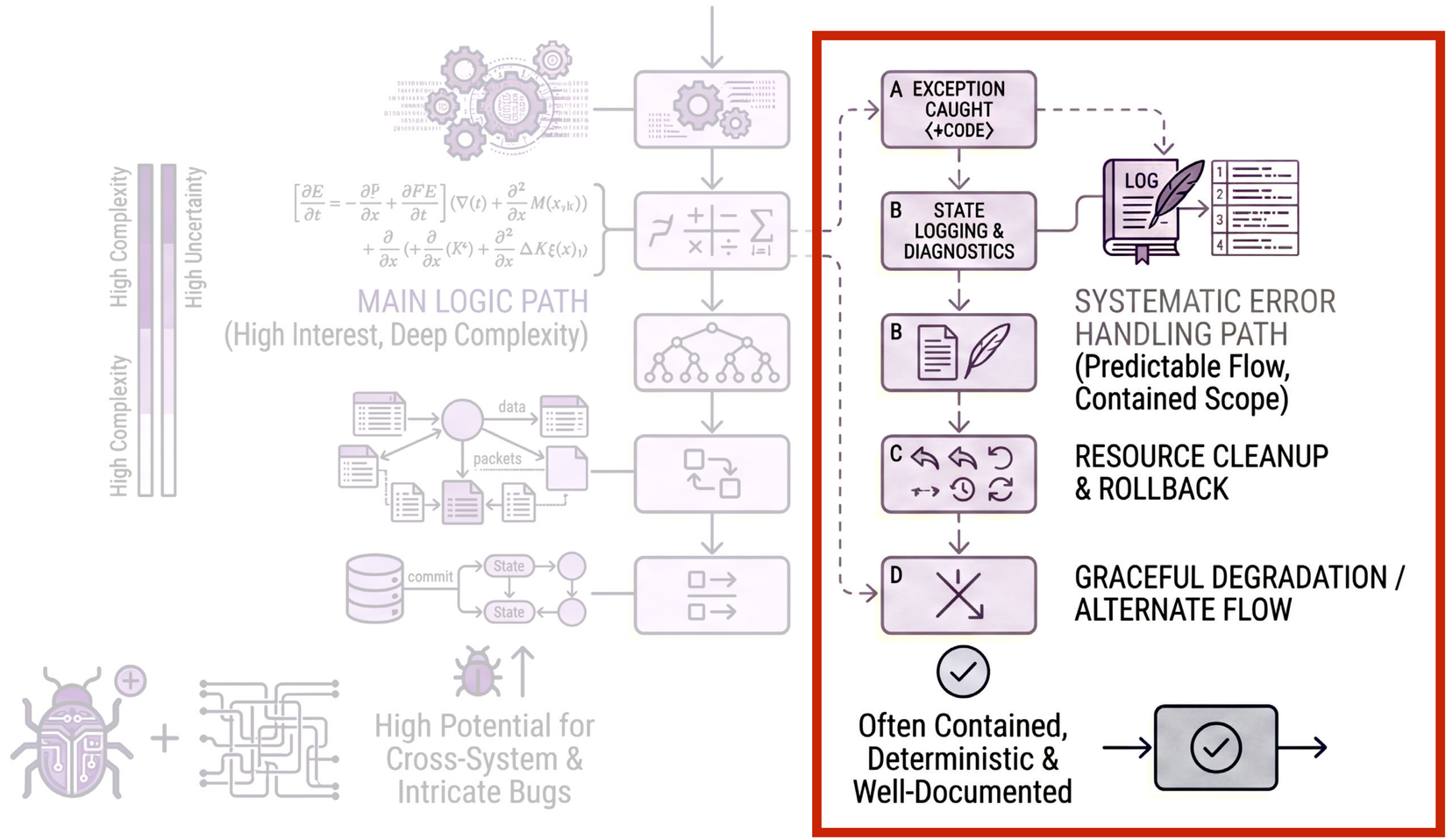
in as little code as possible?

Is All Code Equally Interesting?



Prompt: Visualize that not all code is equally interesting, neither for discovering bugs nor generally. For example, error paths are oftentimes not really interesting! Make sure that this is simple in design, not busy, has somewhat little text and it is not overly colorful. Use #643A8D as the main color and #FFFFFF for the background.

Is All Code Equally Interesting?



Is this code really equally interesting?

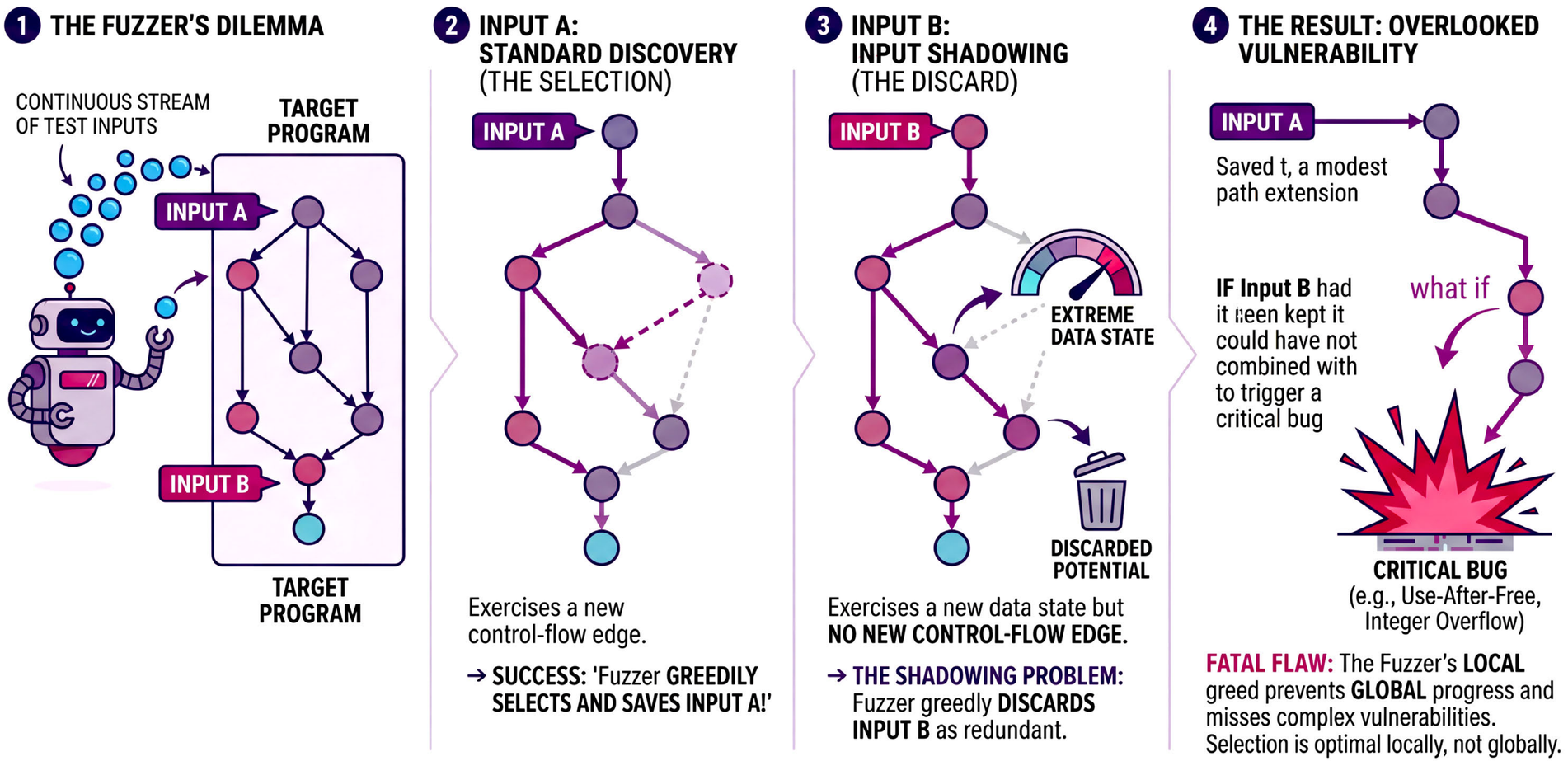
Prompt: Visualize that not all code is equally interesting, neither for discovering bugs nor generally. For example, error paths are oftentimes not really interesting! Make sure that this is simple in design, not busy, has somewhat little text and it is not overly colorful. Use #643A8D as the main color and #FFFFFF for the background.

Coverage-guided Fuzzers are Just Greedy

- By design, coverage-guided fuzzers are greedy algorithms
  - We pick the the first input that triggers new code coverage
  - That is, adding it to the set of inputs instead of the previously-tested inputs because it gives us the locally optima of total covered code
    (for all currently reachable code, constrained by the mutation operations)
- But, fuzzing is not an optimal substructure problem!
  - Achieving the maximum coverage, given a set of mutation operations, with a set of inputs on the first set of parts a program does not mean that we can achieve the maximum coverage on the entire program with those inputs!

Are we instructing our fuzzers to throw away the keys to the next bug?! 😵‍💫

Input Shadowing



Prompt: Visualize the fatal flaw of edge-driven selection: The problem of "Input Shadowing", namely that fuzzers act greedily. If an input exercises a new data state, but *doesn't* flip a completely new control-flow edge, then the fuzzer discards it, but that is not necessarily optimal. Use #643A8D as the main color, #AA3E6F as the first accent color, and then colors in the same color style. Use complete white, #FFFFFF, for the background.

Traditional Code Coverage

- Traditional code coverage checks if we went into every room in a building, but it entirely ignores what we brought into the room or took from it.
- Coverage-guidance does not prevent us from entering the room again, but we cannot leave things in the room or take things from it. (we reset to our previous save point)
- We will never be able to bring all the items needed to break the vault door and enter the vault. That is, we can never satisfy the preconditions to enter the vault room. (i.e., trigger the bug)

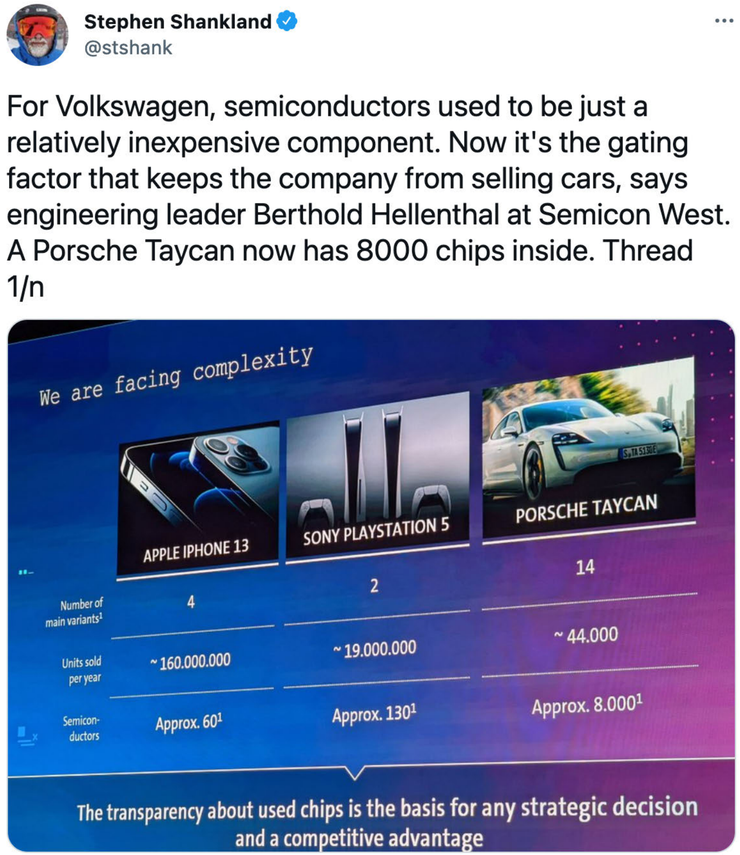
Bugs Continue to Survive Extended Continuous Fuzzing

- WingFuzz (Data Coverage for Guided Fuzzing, 2024)
  *"we have discovered 28 previously-unknown bugs on OSS-Fuzz projects that were **well-fuzzed using code coverage**."*

- StorFuzz (Data Diversity to Overcome Fuzzing Plateaus, 2026)
  *"[...] leading to the discovery of 50 new bugs in 7 OSS-Fuzz projects, like VLC and PHP, with some bugs having been present in the code for **14 years.**"*

"Bugs that survive the heat of continuous fuzzing", Antonio Morales, https://github.blog/security/vulnerability-research/bugs-that-survive-the-heat-of-continuous-fuzzing/
"Data Coverage for Guided Fuzzing", Wang et al., USENIX Security 2024, https://www.usenix.org/conference/usenixsecurity24/presentation/wang-mingzhe
"StorFuzz: Using Data Diversity to Overcome Fuzzing Plateaus", Weiß et al, ICSE 2026, https://softsec.link/icse26.storfuzz

Real-world Software Systems are Interconnected (and Stateful)



- Average mobile app communicates with ~9 different Internet services
  - Also talks to other apps, the OS, etc.
- Average car has 30–50 networked computers, high-end cars have 100+, they talk to each other
  - Most run normal software on fully-fledged normal CPUs: An ARM Cortex M0+ costs €0.66 wholesale

Code coverage is misleading for these systems!

Stateful Systems

- A stateful system is a system whose behavior depends on code **and data** (often previous input data)
- A core issue in distributed/networked systems
  - And beyond: e.g., games: Position of the player and enemies
- Bugs depend on state
  - Assuming a state and accessing unassigned variables, pointers etc.

```
if (!builtin_call_pa && curr_scope != top) {
  update_call_arg_offset(ctx, ret_type, 1);
}
```

Stateful Systems

Bugs can also hide behind stateful requirements

```
detect_attributes(file);
switch (file->type) {
case TYPE_A:
  node = file->getFirstNode();
  for (; node != nullptr; node = node->next()) {
    if (node->attributes & ATTRIBUTE_B) {
      next_node = node->next();
      node->parsed = true;
      // ...
    }
    //...
  }
  break;
  // other cases
}
```

How can we find these bugs?

Stateful Fuzzing

- Modern fuzzers (AFL++, LibAFL, etc.) **still aim to max code coverage**
- Stateful: **Same code behaves differently for different data**
  - Code coverage is the same, but **behavior with different data is (very) different**

```
char buf[10];
for (int i = 0; i < input_len; ++i) {
  buf[i] = input[i];
}
```

Same code coverage for:

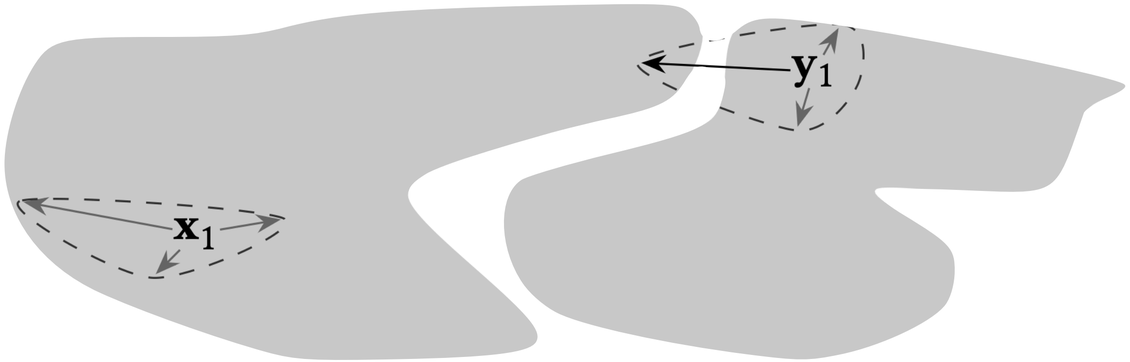
- `input_len < 10`
- `input len >= 10`

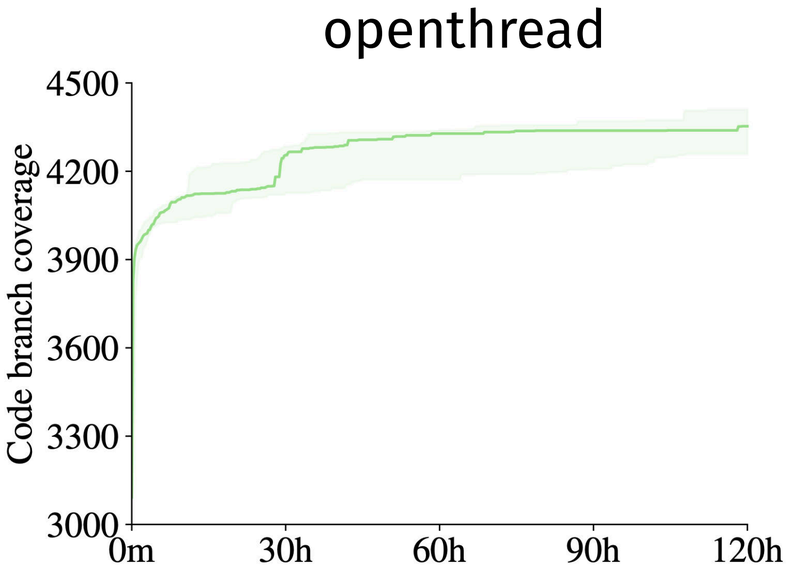
But >= 10 can (maybe) hijack control flow

- Some fuzzers try to incorporate state, but typically require manual definition of state variables or are restricted to specific types of how state can be encoded ⇒ No known general approach

State of the Art Fuzzers Plateau

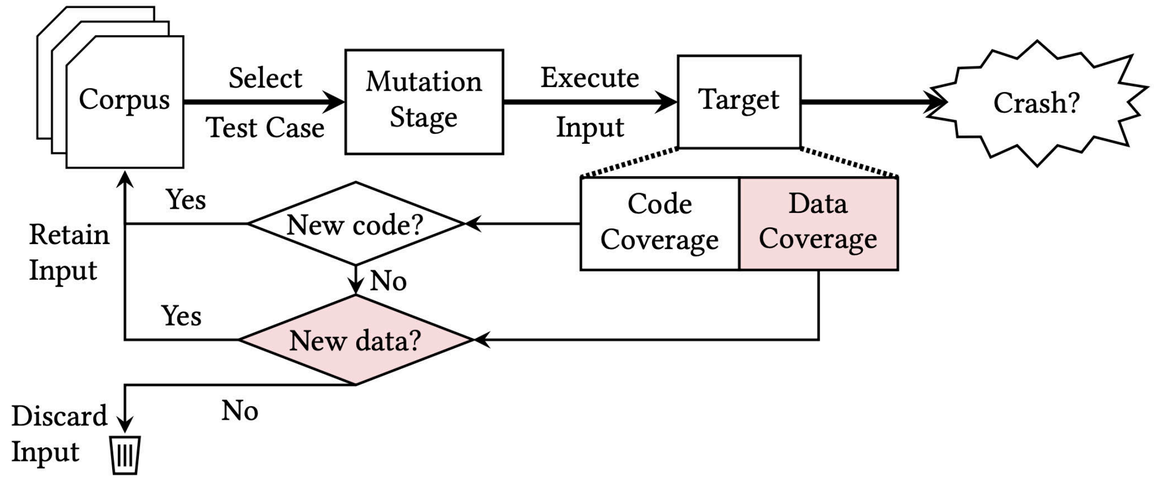
- Modern fuzzers plateau after ~48h in code coverage and do not progress anymore
  - They only keep one input sample for each code coverage equivalence class, but distance to next class might be (too) large
  - Different states may also have the same code coverage
- Fuzzers plateauing and being able to cover state are connected!





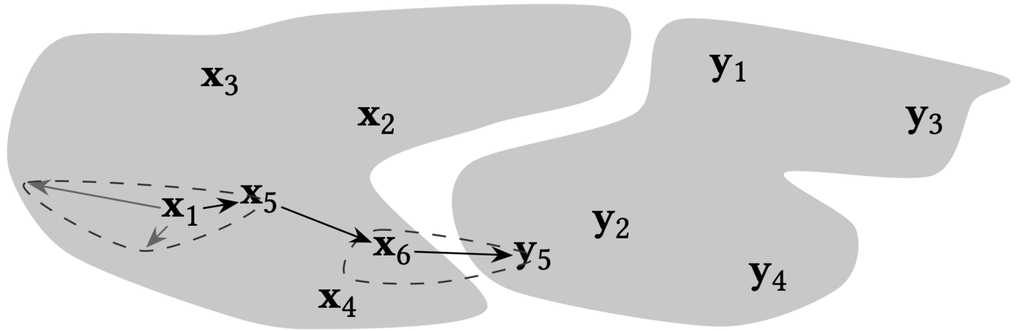
StorFuzz: Data Coverage

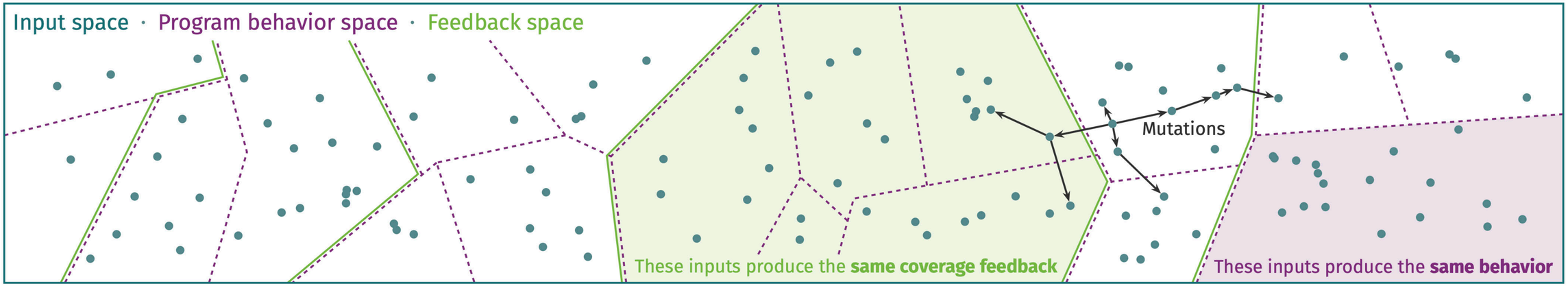
- No known general approach to define what state is
  - We can over-approximate: state must be encoded in data
- New coverage measure: data coverage
  - Instrument and record memory stores (location : abstracted values)
  - Focus on long-lived data
  - Ignore constants
  - Heuristically skip pointers



StorFuzz: Data Diversity

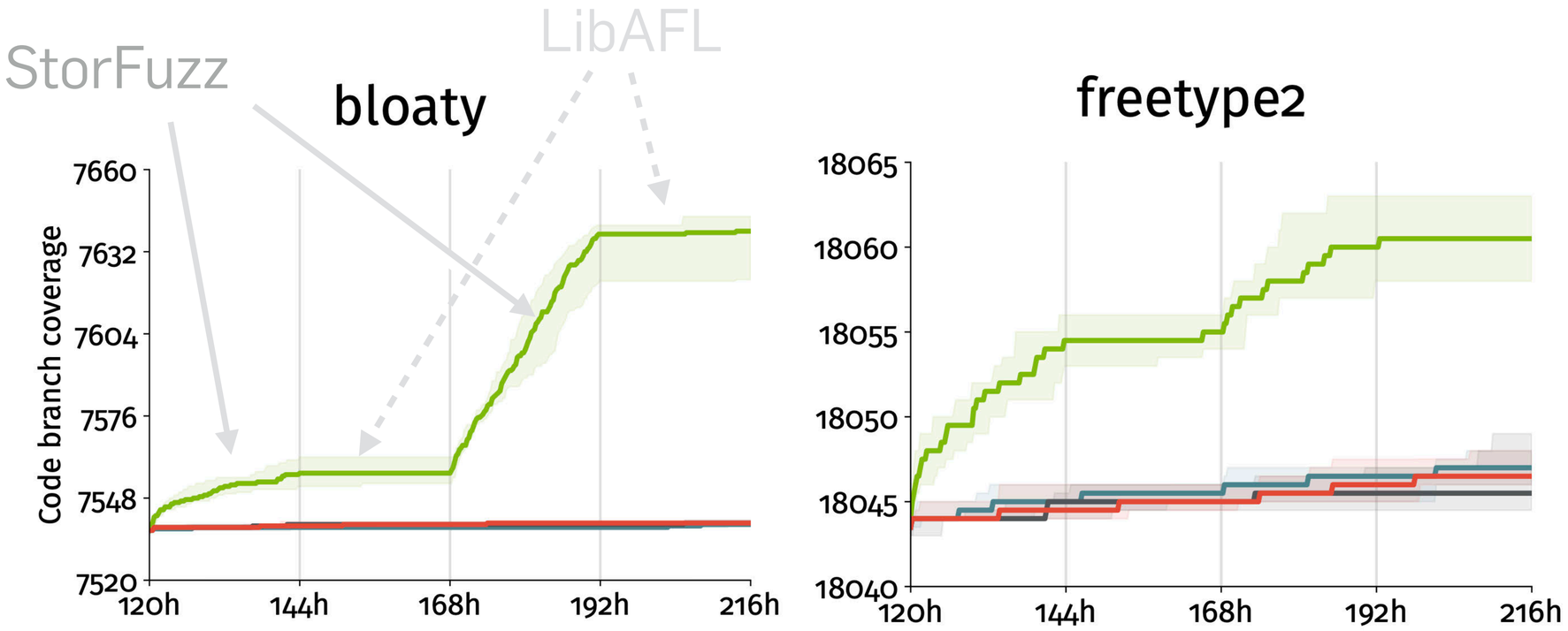
- StorFuzz considers data coverage for the equivalence classes, further partitioning them
  - This allows us to break out of plateaus
  - Enables stateful fuzzing with low overhead





StorFuzz: Code Coverage Improvements

but without actually aiming to improve code coverage!



LibAFL / Restarting LibAFL / DDFuzz+LibAFL / StorFuzz+LibAFL

StorFuzz: Bugs and Vulnerabilities?

- More code coverage is great, but do we find more bugs?
- We looked at projects in Google's OSS-Fuzz
  - Targets typically fuzzed extensively for multiple years (the real problem)
    - e.g., VLC is being fuzzed for 4+ years now as part of OSS-Fuzz
- We discovered 50 new, previously unknown vulnerabilities (and rediscovered at least 20 unresolved bugs)
  - One vulnerability in VLC was introduced in 2011, but no other approach was able to find it beforehand
  - For another vulnerability, OSS-Fuzz actually the same code, but couldn't reach the states to trigger the bugs

Understanding Program State

- Combining
  - Data coverage, our simple metric to measure data diversity
  - Observed input/output behavior with StorFuzz (at higher data diversity)
- We can (better) learn an **under-approximate understanding** of how a program implements state
  - Under-approximate because we rely on fuzzing, might not see everything
  - Definition of I/O behavior restricts type of states we learn about
- Generally more work needed to better understand program state automatically! (yes, LLMs can definitely be helpful here)

Code Coverage Just Won't Suffice in the Future

- There is a clear trend and push toward memory-safe languages for new code and even rewriting old code in them
  - Substantial code is now also AI-generated, much of which is in (largely) memory-safe languages, like JavaScript, Python, etc.
  - The future of vulnerability discovery lies in (business) logic bugs and (protocol) state confusion
- Semantic and logical vulnerabilities heavily depend on state
  - Thus, code coverage won't suffice anymore in the future
  - They will also require us to properly adopt bug oracles

Bug-based Benchmarks?

- "In Google's FuzzBench platform, we find that the outcome of coverage-based evaluation more strongly agrees with the outcome of a bug-based evaluation than an independent bug-based evaluation itself."
  "In Bugs We Trust? On Measuring the Randomness of a Fuzzer Benchmarking Outcome", Madadi et al., FSE 2026 (to appear)

- "The fuzzer best at achieving coverage, may not be best at finding bugs."
  "On the reliability of coverage-based fuzzer benchmarking", Böhme et al., ICSE 2022

Maybe our bug-based benchmarks are just not good? 🧐

"In Bugs We Trust? On Measuring the Randomness of a Fuzzer Benchmarking Outcome", Madadi et al., FSE 2026 (to appear), DOI: 10.1145/3797112
"On the reliability of coverage-based fuzzer benchmarking", Böhme et al., ICSE 2022, DOI: 10.1145/3510003.3510230

But Doesn't AI Make Fuzzing Unnecessary Anyways?

No.

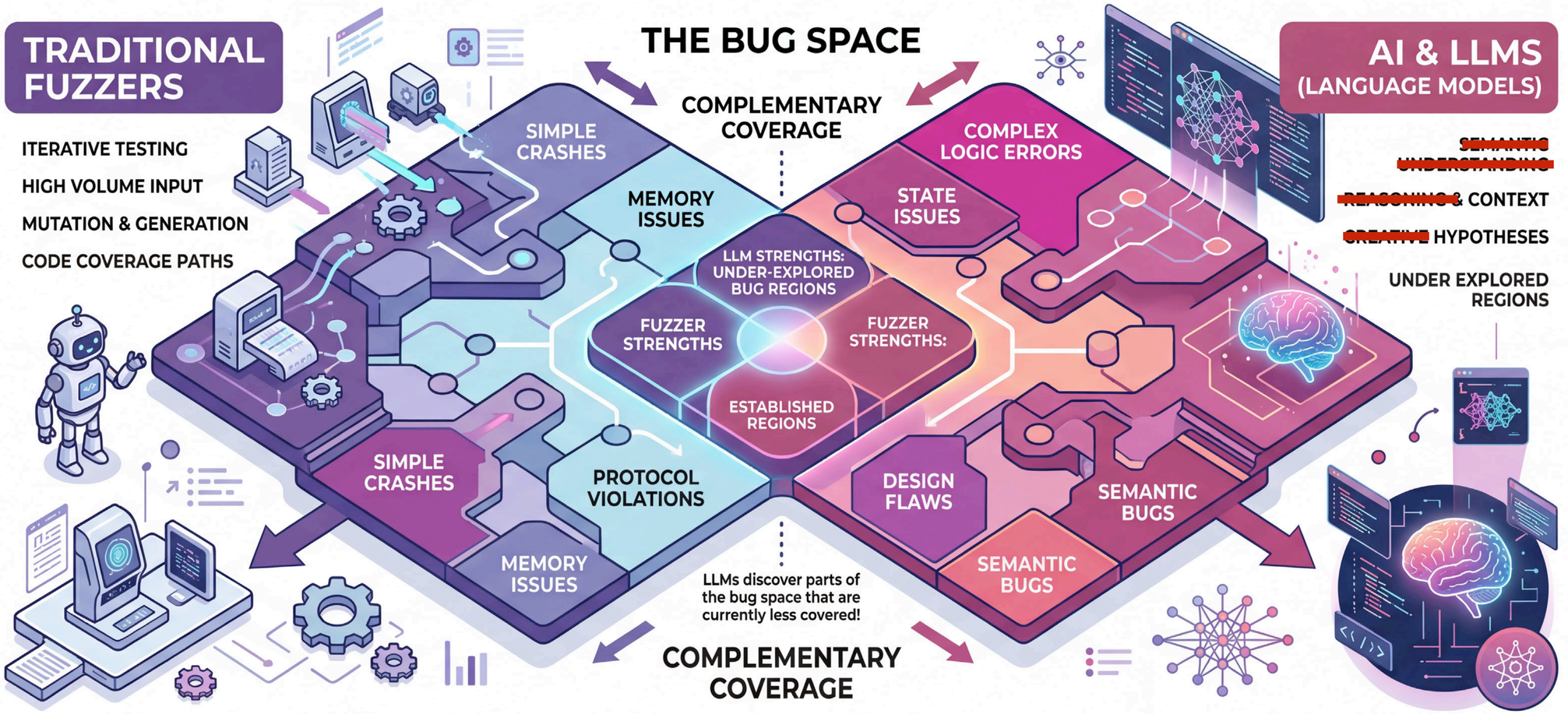
Some AI companies want
you to believe it though.

Bug Space

- The **bug space** is clearly a subspace of all program behavior
- AI finds bugs in programs that we missed with fuzzing, why?
  a) AI is fundamentally better and we should abandon fuzzing
  b) Fuzzing could find the bugs, but we didn't apply it properly

From all I've seen, b) appears to be true.

Exploring The Bug Space



Prompt: Visualize that fuzzers and AI explore/cover different parts of the bug space. That is, they can explore different parts of the same space. LLMs currently work well because they explore parts of the bug space that we haven't covered much! Use #643A8D as the main color, #AA3E6F as the first accent color, and then colors in the same color style. Use complete white (#FFFFFF) for the background.

Anthropic's Claude Opus 4.6

- *"Let me check if maybe the checks are incomplete or there's another code path. Let me look at the other caller in gdevpsfx.c … Aha! This is very interesting! In gdevpsfx.c, the call to gs_type1_blend at line 292 does NOT have the bounds checking that was added in gstype1.c."*
  Sounds like a harness/exploration issue combined with maybe missing data/value coverage?

- *"CGIF had implicitly assumed that the compressed size of a compressed string would always be less than the uncompressed size—something that is almost always true."*
  Sounds awfully like a data/value coverage issue?

Bug Space

- The **bug space** is clearly a subspace of all program behavior
- AI finds bugs in programs that we missed with fuzzing, why?
  a) AI is fundamentally better and we should abandon fuzzing
  b) Fuzzing could find the bugs, but we didn't apply it properly

However, this does not mean fuzzing will be more effective or efficient at finding them!

(luckily, our budget is high because AI is not particularly resource efficient)

Takeaways

- Fuzzing is no longer bottlenecked by compute. It is bottlenecked by our selection metrics.
- Code coverage ain't enough for a post-memory corruption world. Embrace precise and problem-targeted data and value coverage!
- Fuzzing must progress toward a proper understanding of implementations and statefulness. Otherwise, it will play little to no role in developing safe, reliable, and trustworthy software systems.

Questions?

kevin.borgolte@rub.de
https://softsec.rub.de

Our research is supported by:

CASA Securing the Digital Society

DFG Deutsche Forschungsgemeinschaft German Research Foundation

WWTF Vienna Science and Technology Fund

Internet Society Foundation

ICANN