DY Fuzzing: Formal Dolev-Yao Models Meet Protocol Fuzz Testing

Lucca Hirschi, **Inria Nancy**

joint work with Steve Kremer and Max Ammann

June 28th 2023 @ JN GDR Sécurité, Paris

Secure Cryptographic Protocols

Cryptographic Protocols

Informal definition

concurrent program relying on cryptography to secure communications

Examples: TLS, EMV (credit cards), RFID, e-voting, mobile com., etc.

Cryptographic Protocols

Informal definition

concurrent program relying on cryptography to secure communications

Examples: TLS, EMV (credit cards), RFID, e-voting, mobile com., etc.

- Notoriously difficult to design and deploy securely
- Loads of failure stories: attacks, fixes, attacks, fixes, attacks, etc.

Cryptographic Protocols

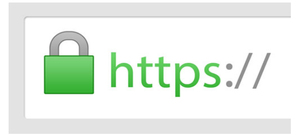
Informal definition

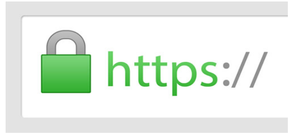
concurrent program relying on cryptography to secure communications

Examples: TLS, EMV (credit cards), RFID, e-voting, mobile com., etc.

- Notoriously difficult to design and deploy securely
- Loads of failure stories: attacks, fixes, attacks, fixes, attacks, etc.

☞ What can we do today to avoid such failures in the future?

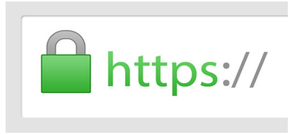
Retrospective of TLS Failures  2014-2022

Retrospective of TLS Failures  2014-2022

HeartBleed

Apple’s GotoFail

CloudBleed

Retrospective of TLS Failures  2014-2022

HeartBleed

Apple’s GotoFail

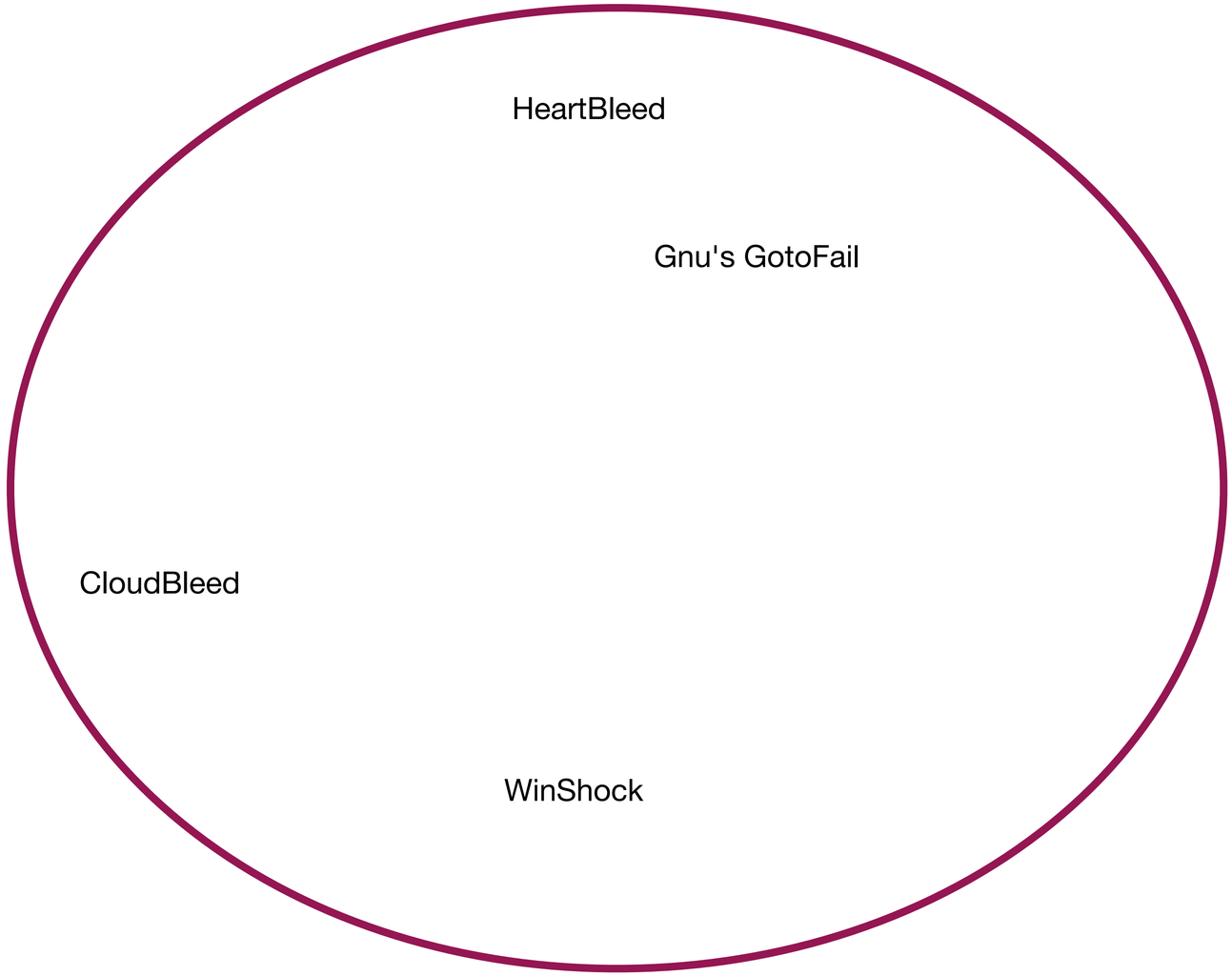
CVE-2022-25640

FREAK

CloudBleed

3SHAKE

Retrospective of TLS Failures https:// 2014-2022



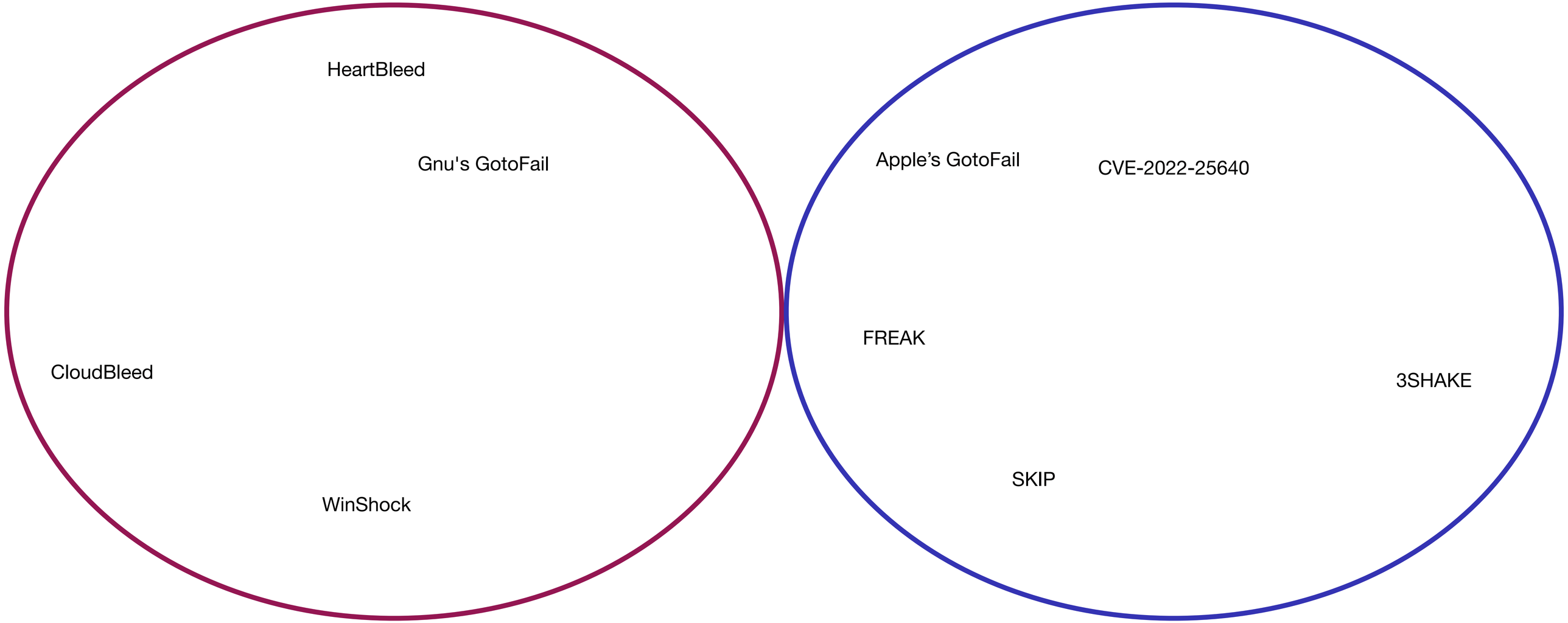
Apple’s GotoFail

CVE-2022-25640

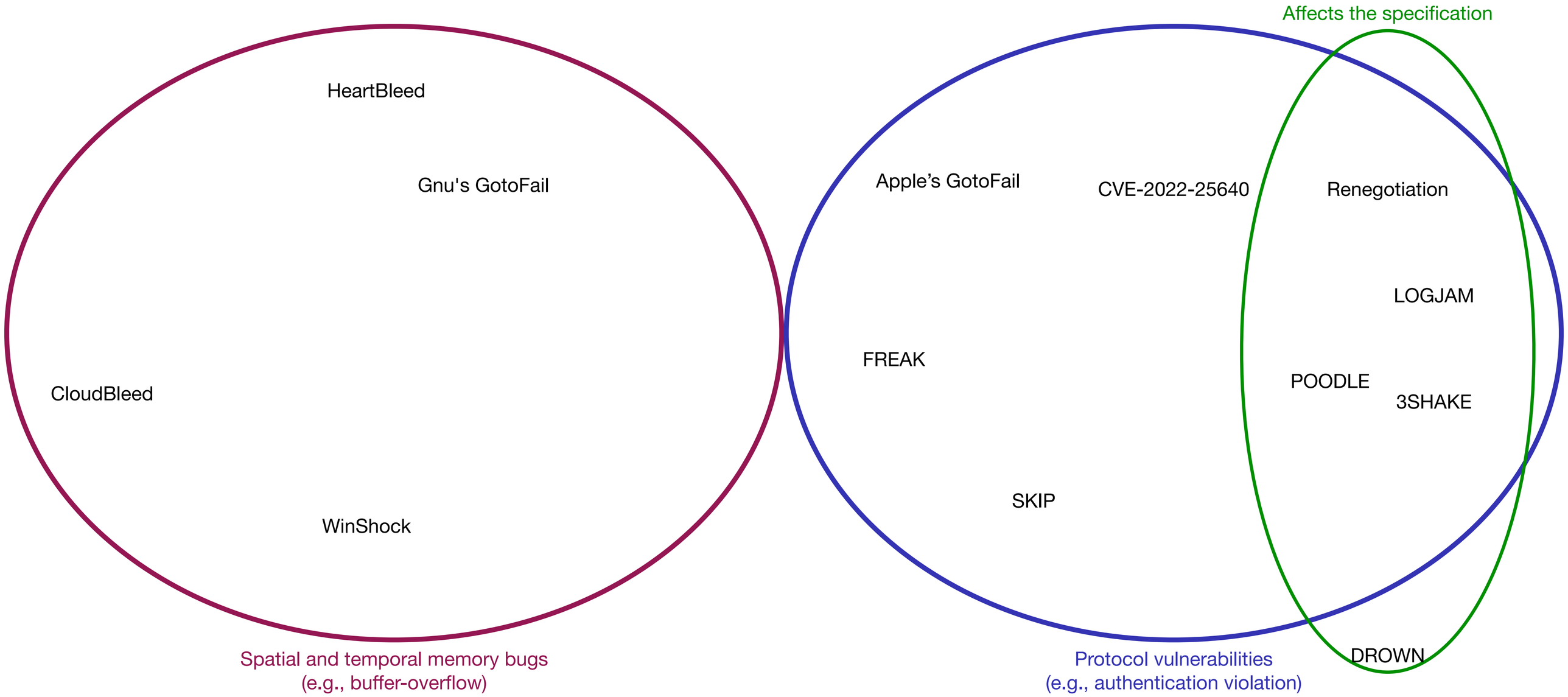
FREAK

3SHAKE

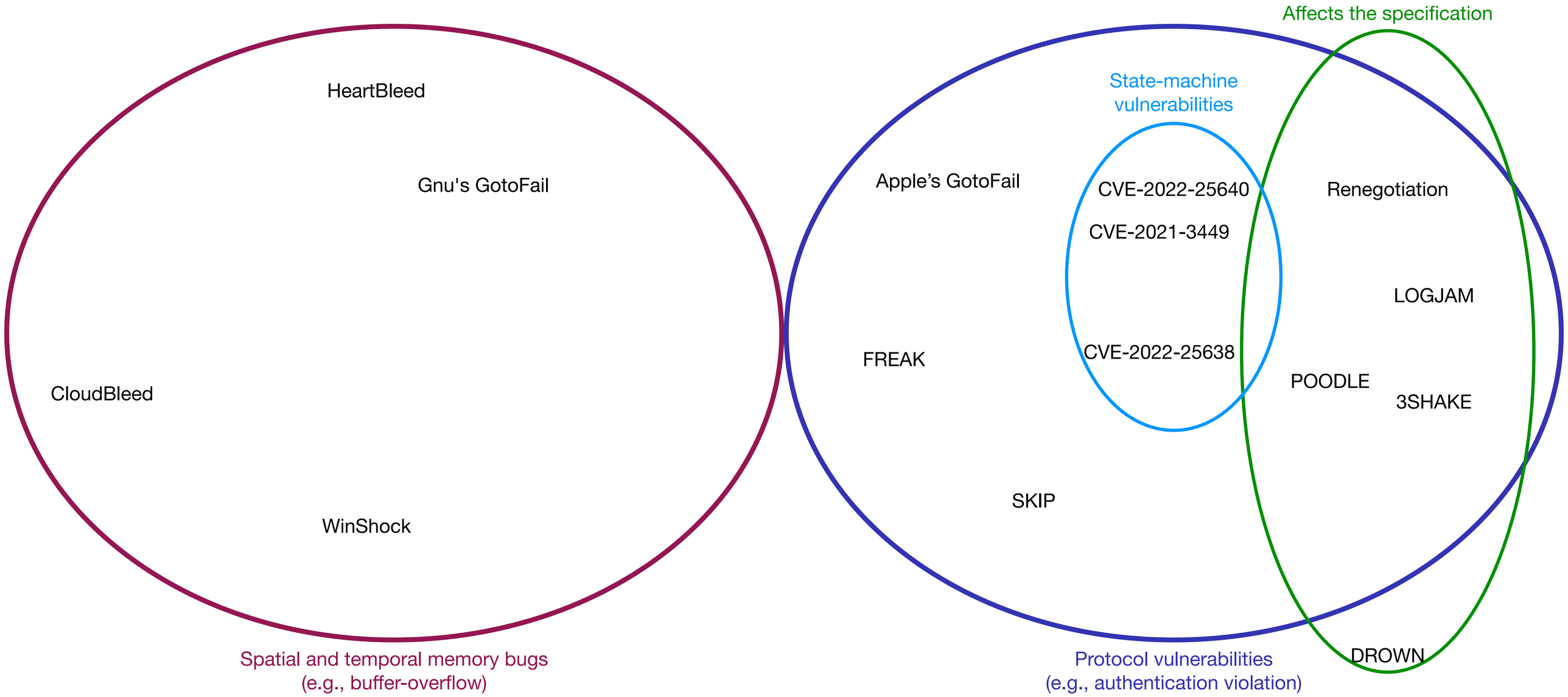
Retrospective of TLS Failures https:// 2014-2022



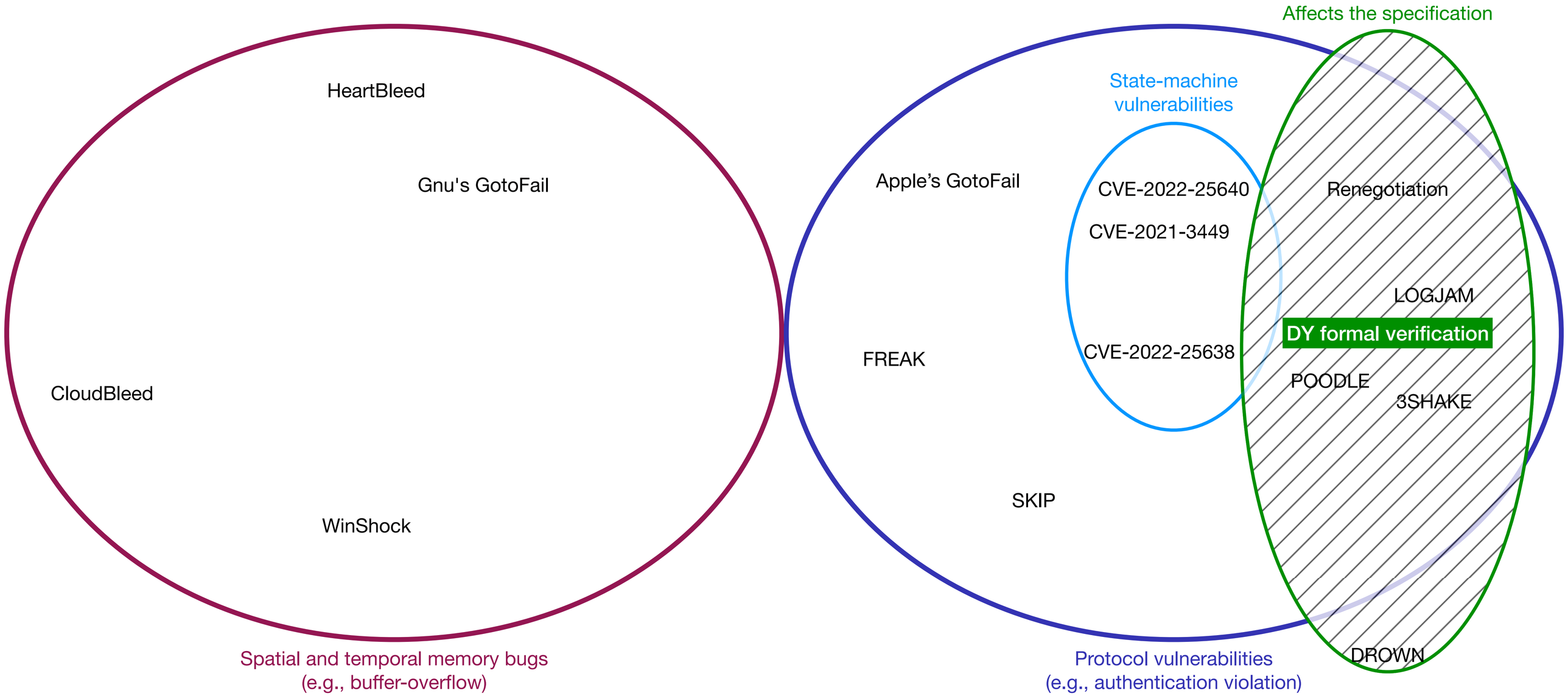
Retrospective of TLS Failures https:// 2014-2022



Retrospective of TLS Failures https:// 2014-2022



Retrospective of TLS Failures https:// 2014-2022



Dolev-Yao Formal Verification

Dolev-Yao Formal Verification

- Formal models for analyzing cryptographic protocols (e.g., applied-π calculus)

Dolev-Yao Formal Verification

- Formal models for analyzing cryptographic protocols (e.g., applied-π calculus)
- Threat model:

  i. active adversary controlling the network (**intercept, modify, inject**)
  ii. is able to **use cryptography**
  iii. cryptography as **black-box** (attacker's interface = functionality)

Dolev-Yao Formal Verification

- Formal models for analyzing cryptographic protocols (e.g., applied-π calculus)
- Threat model:

  i. active adversary controlling the network (**intercept, modify, inject**)
  ii. is able to **use cryptography**
  iii. cryptography as **black-box** (attacker's interface = functionality)

- « Messages as formal terms » paradigm:

  E.g., symmetric encryption:
  - **function symbols** enc/2, dec/2 and
  - **equation** dec(enc(m,k),k) = m

Dolev-Yao Formal Verification

- Formal models for analyzing cryptographic protocols (e.g., applied-π calculus)
- Threat model:

  i. active adversary controlling the network **(intercept, modify, inject)**
  ii. is able to **use cryptography**
  iii. cryptography as **black-box** (attacker's interface = functionality)

- « Messages as formal terms » paradigm:

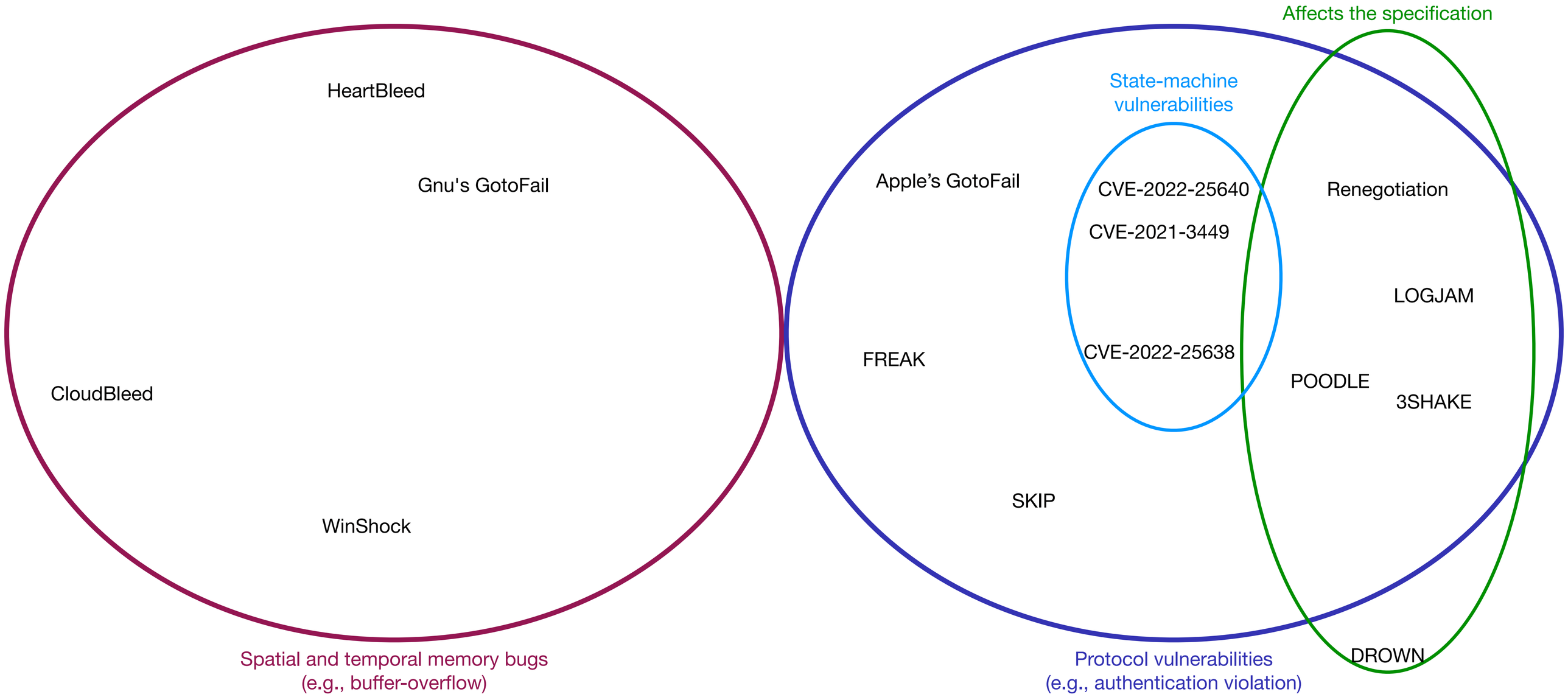
  E.g., symmetric encryption:
  - **function symbols** enc/2, dec/2 and
  - **equation** dec(enc(m,k),k) = m

☞ Find or prove the absence of design-level logical attacks since the 80s

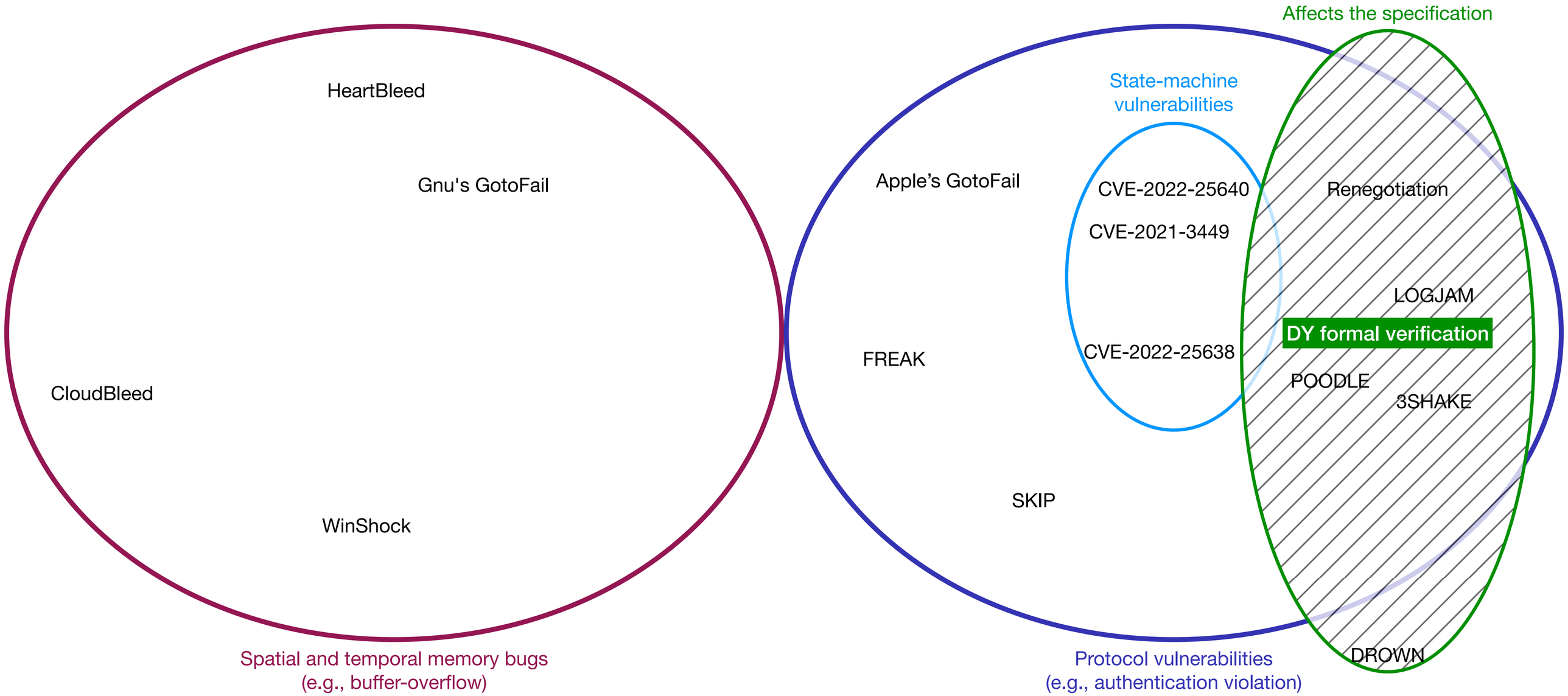
E.g., MITM, downgrade, impersonation, authentication bypass, Unknown Key-Share (UKS), Key Compromise Impersonation (KCI), cross-protocol, and protocol composition attacks, etc.

☞ Limited to specifications, *existing* implementations are out of scope (e.g., OpenSSL)

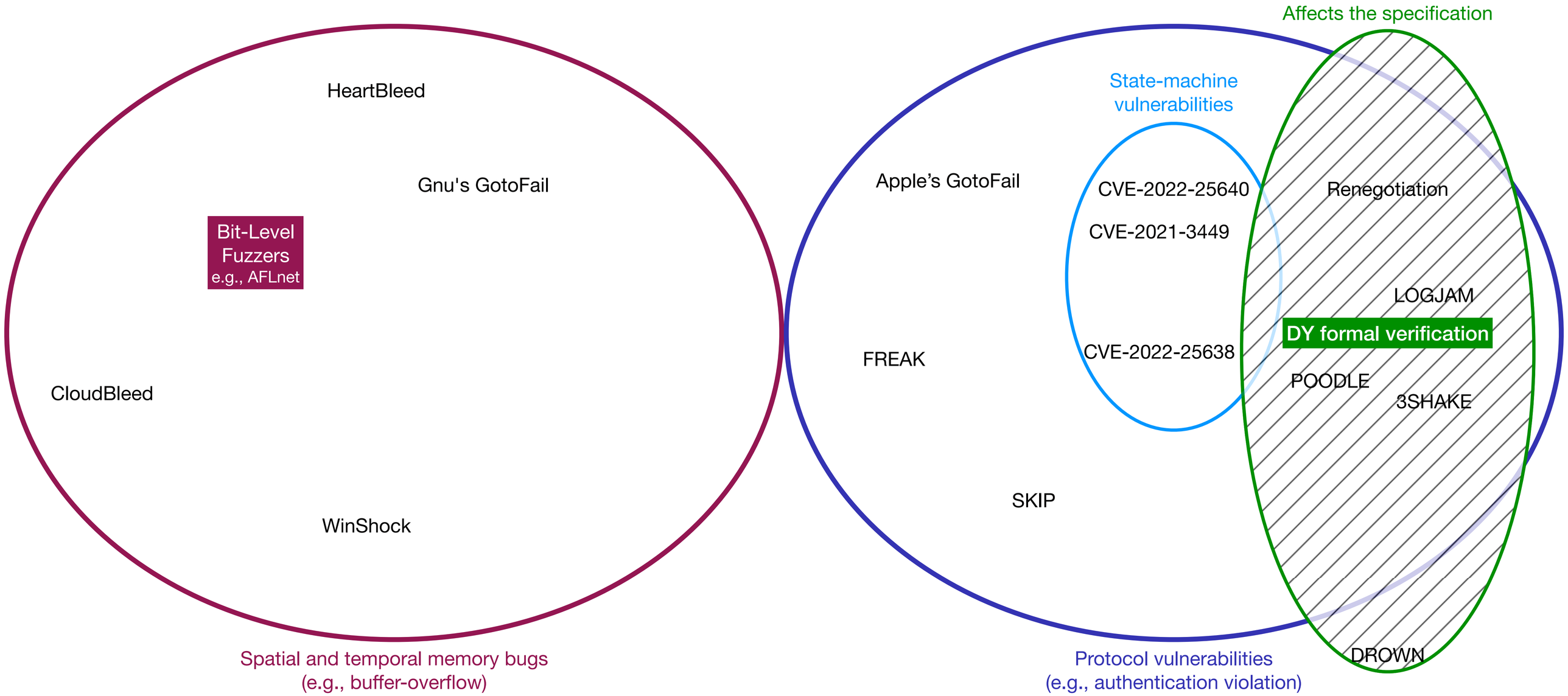
Retrospective of TLS Failures https:// 2014-2022



Retrospective of TLS Failures https:// 2014-2022



Retrospective of TLS Failures https:// 2014-2022

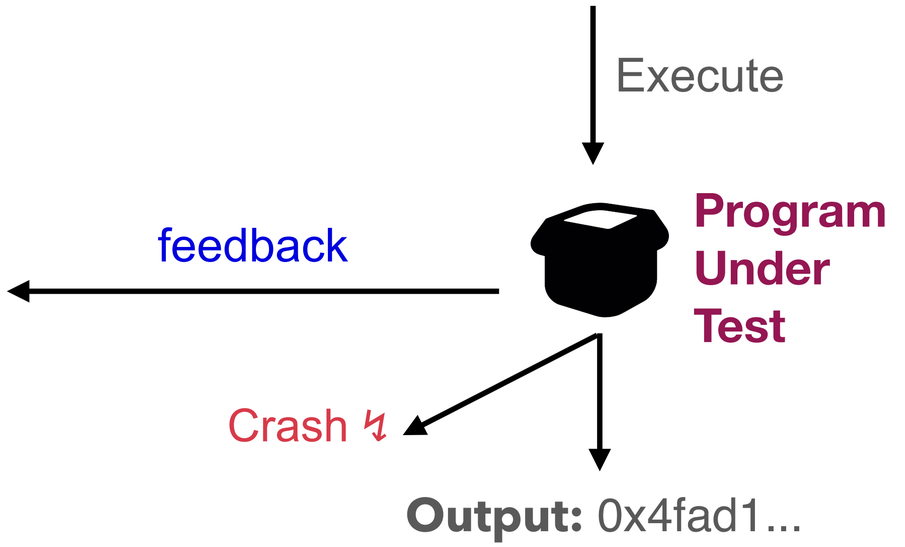


Bit-level fuzzing (AFL-like)

Bit-level fuzzing (AFL-like)

Fuzzing:

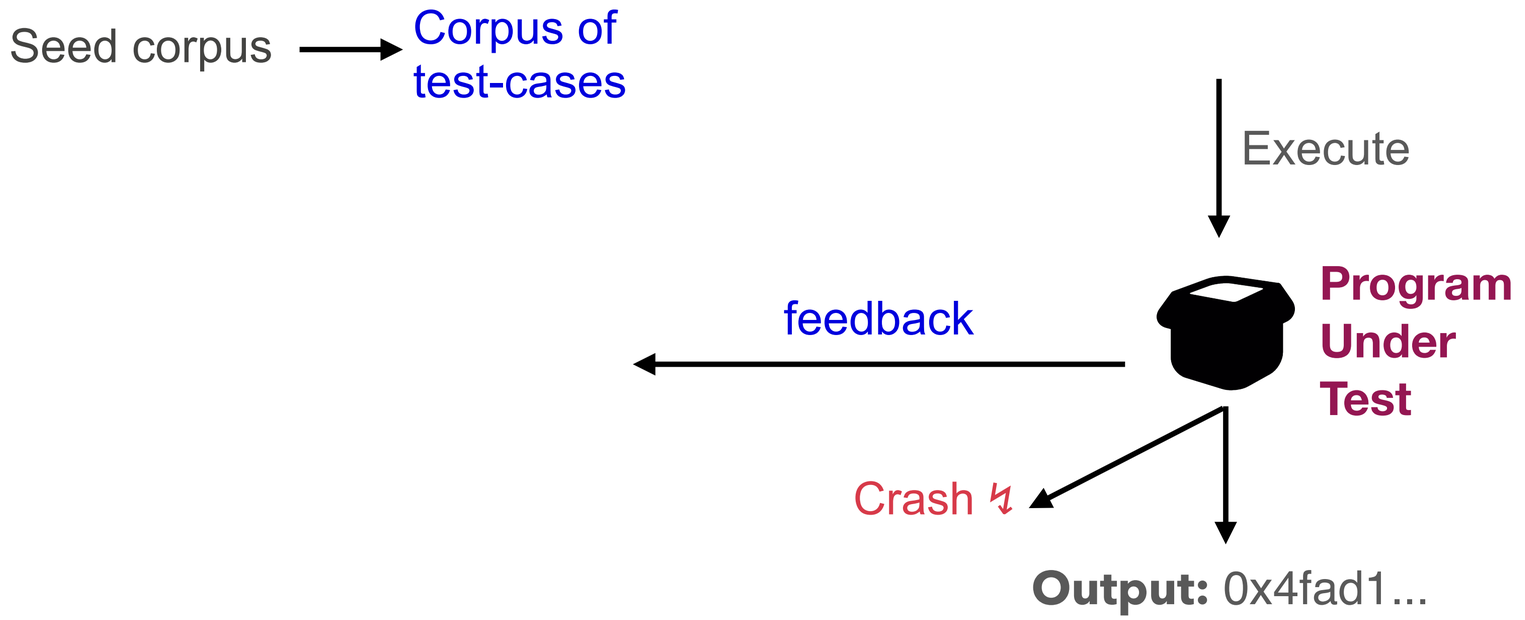
- Instrument the PUT to record feedback



Bit-level fuzzing (AFL-like)

Fuzzing:

- Instrument the PUT to record feedback
- Store a corpus of test-cases

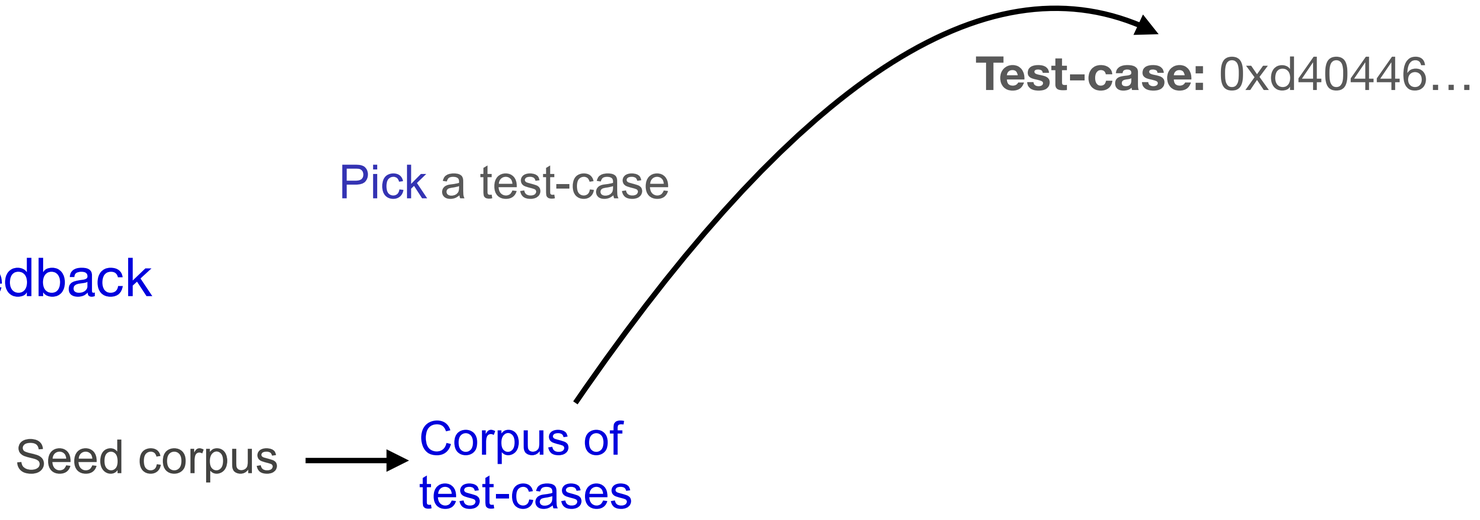


Bit-level fuzzing (AFL-like)

Fuzzing:

- Instrument the PUT to record feedback
- Store a corpus of test-cases

- Fuzzing loop: while true do
  - Pick a test-case



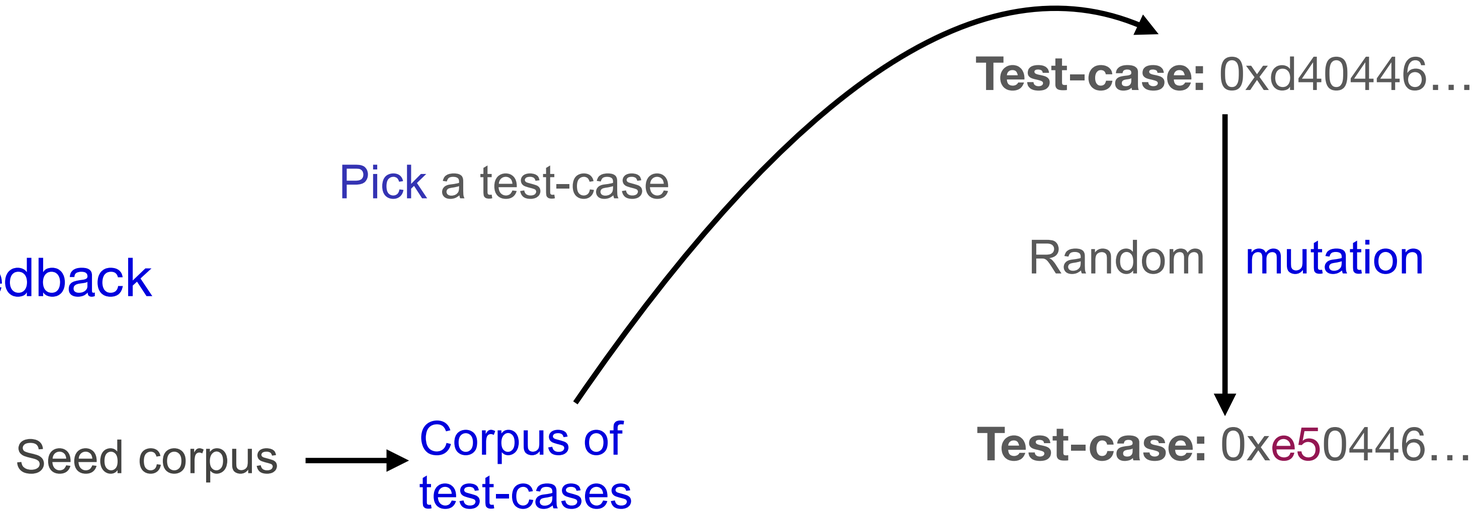
Program Under Test

Bit-level fuzzing (AFL-like)

Fuzzing:

- Instrument the PUT to record feedback
- Store a corpus of test-cases

- Fuzzing loop: while true do
  - Pick a test-case
  - Apply random transformation = mutation



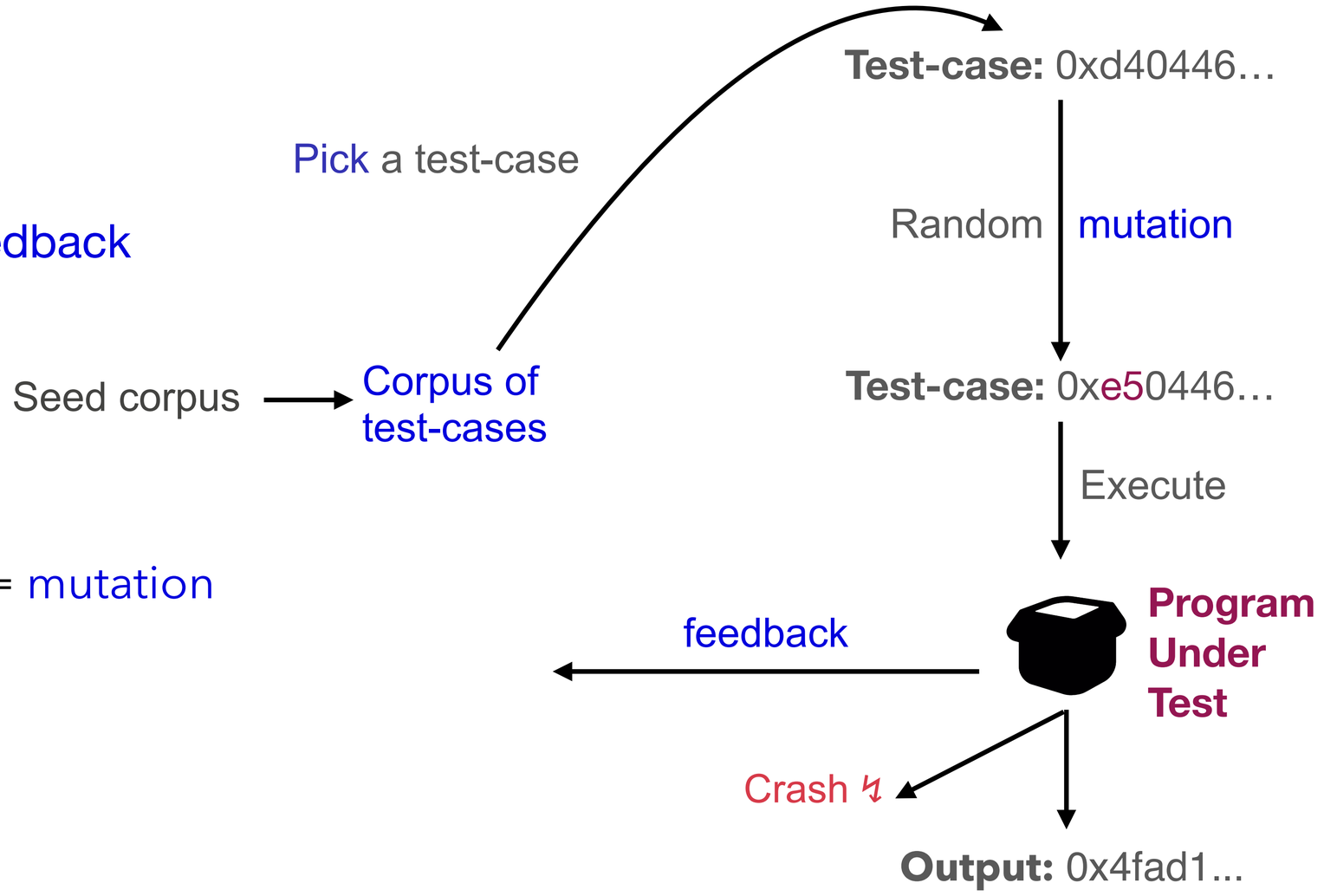
Program Under Test

Bit-level fuzzing (AFL-like)

Fuzzing:

- Instrument the PUT to record feedback
- Store a corpus of test-cases

- Fuzzing loop: while true do
  - Pick a test-case
  - Apply random transformation = mutation
  - Execute + collect feedback

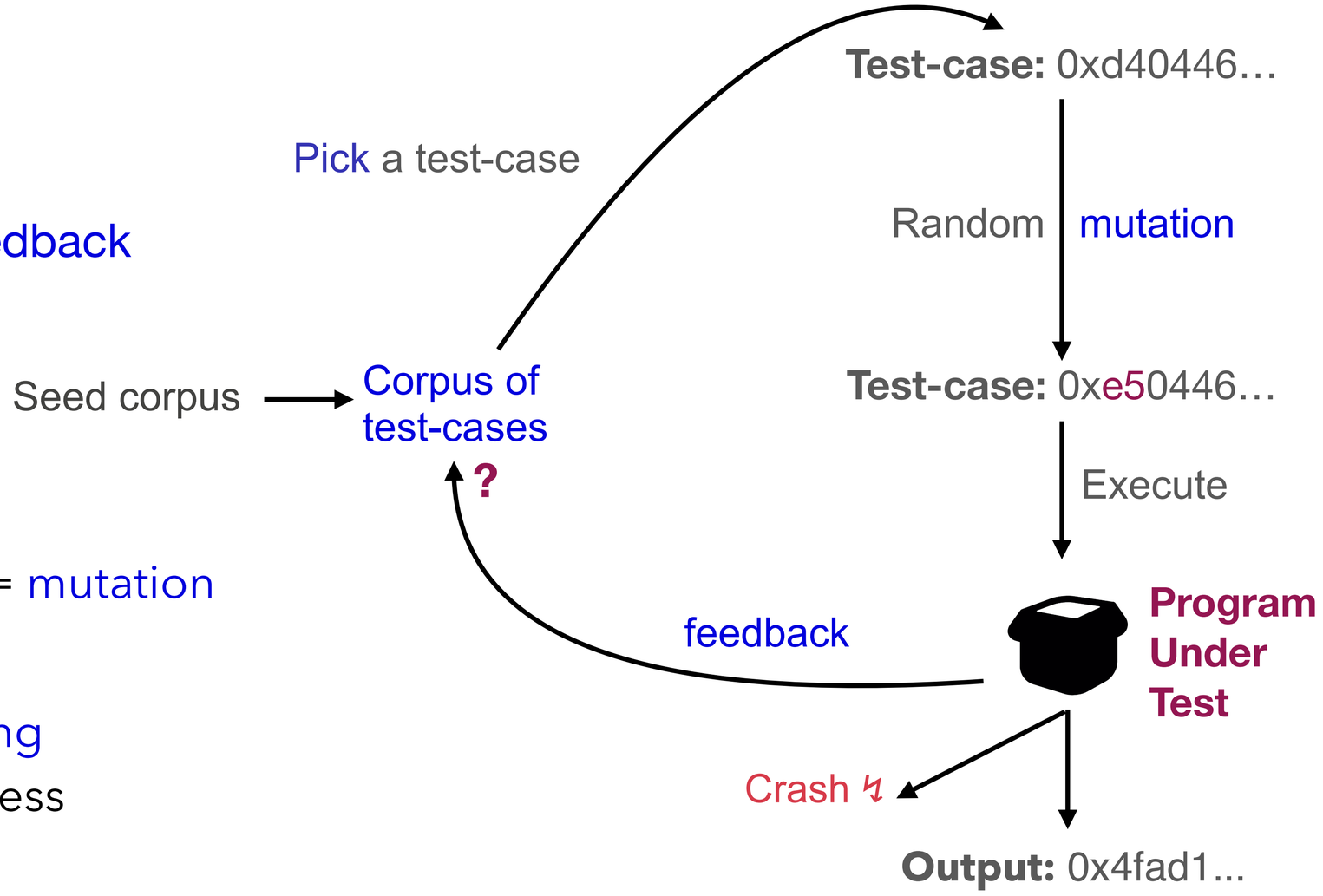


Bit-level fuzzing (AFL-like)

Fuzzing:

- Instrument the PUT to record feedback
- Store a corpus of test-cases

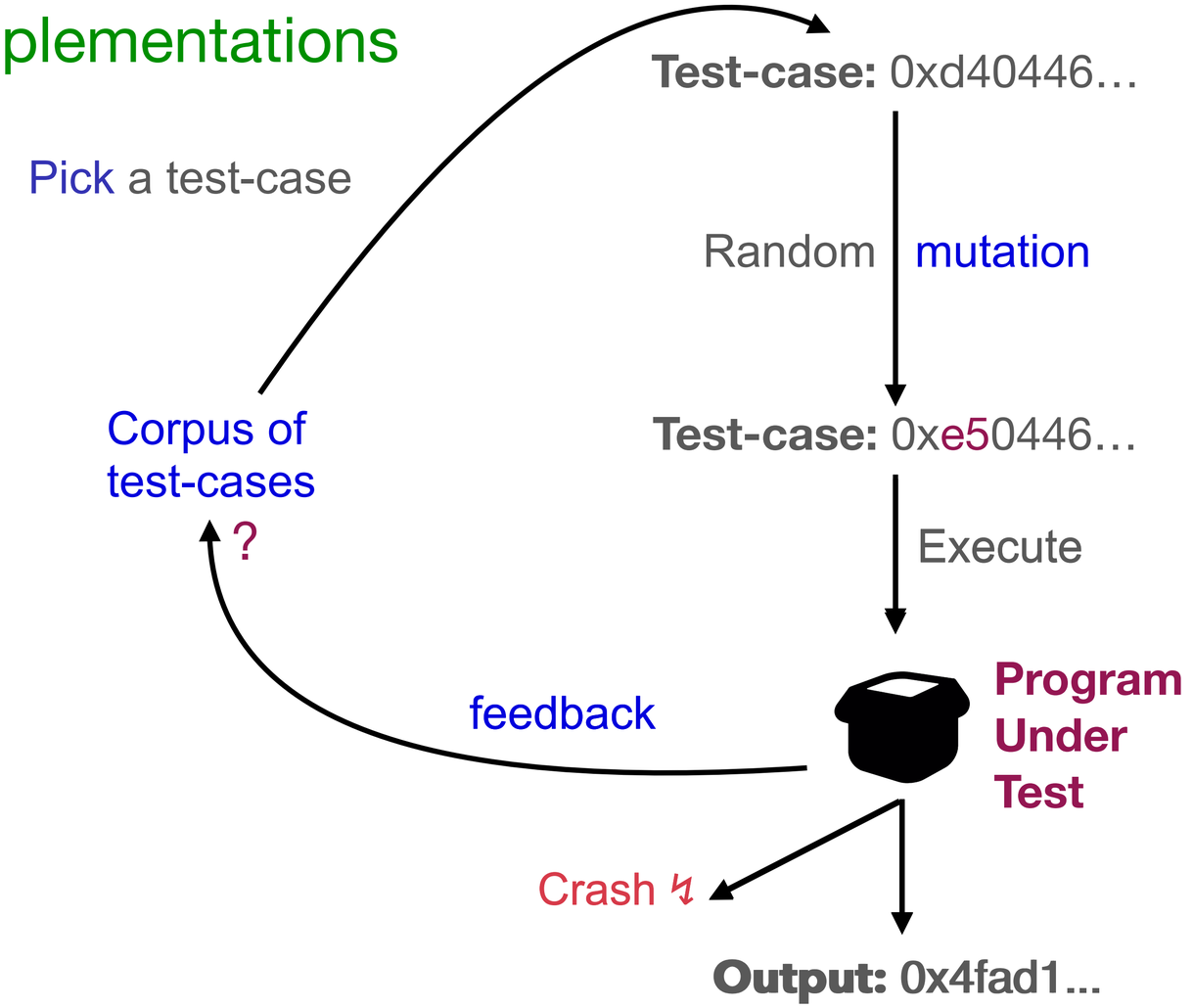
- Fuzzing loop: while true do
  - Pick a test-case
  - Apply random transformation = mutation
  - Execute + collect feedback
  - Add it to the corpus if interesting according to feedback = progress (e.g., new coverage)



Bit-level fuzzing (AFL-like)

☞ Finds memory/crash vulnerabilities in implementations

E.g., buffer-overflow, use after free, RCE, etc.



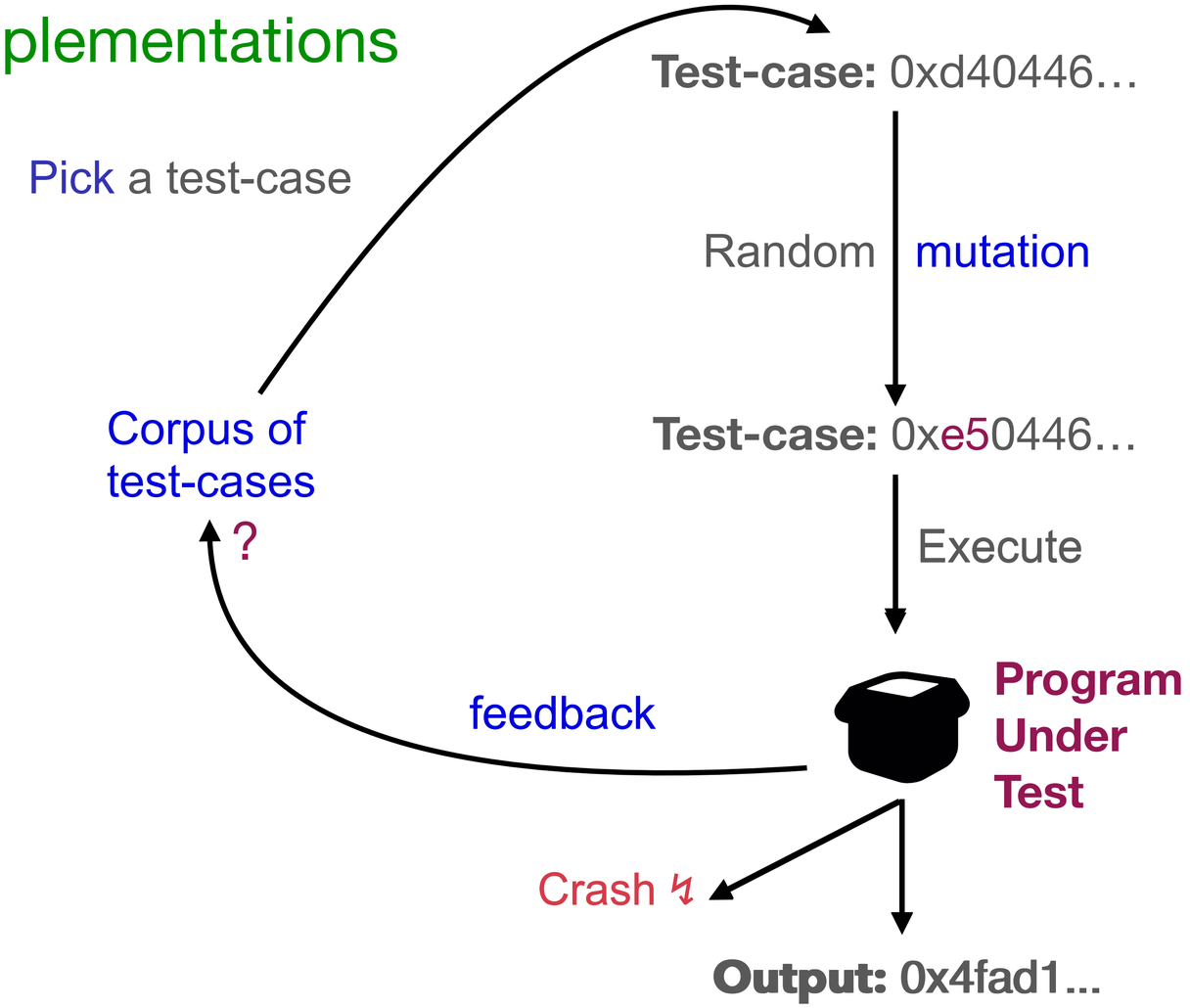
Bit-level fuzzing (AFL-like)

☞ Finds memory/crash vulnerabilities in implementations

E.g., buffer-overflow, use after free, RCE, etc.

☞ Logical attack states are not reached
  ☞ miss some memory vulnerabilities

structural message modification:
e.g., negligible probability of finding a valid encryption through mutations
message flow modification



Bit-level fuzzing (AFL-like)

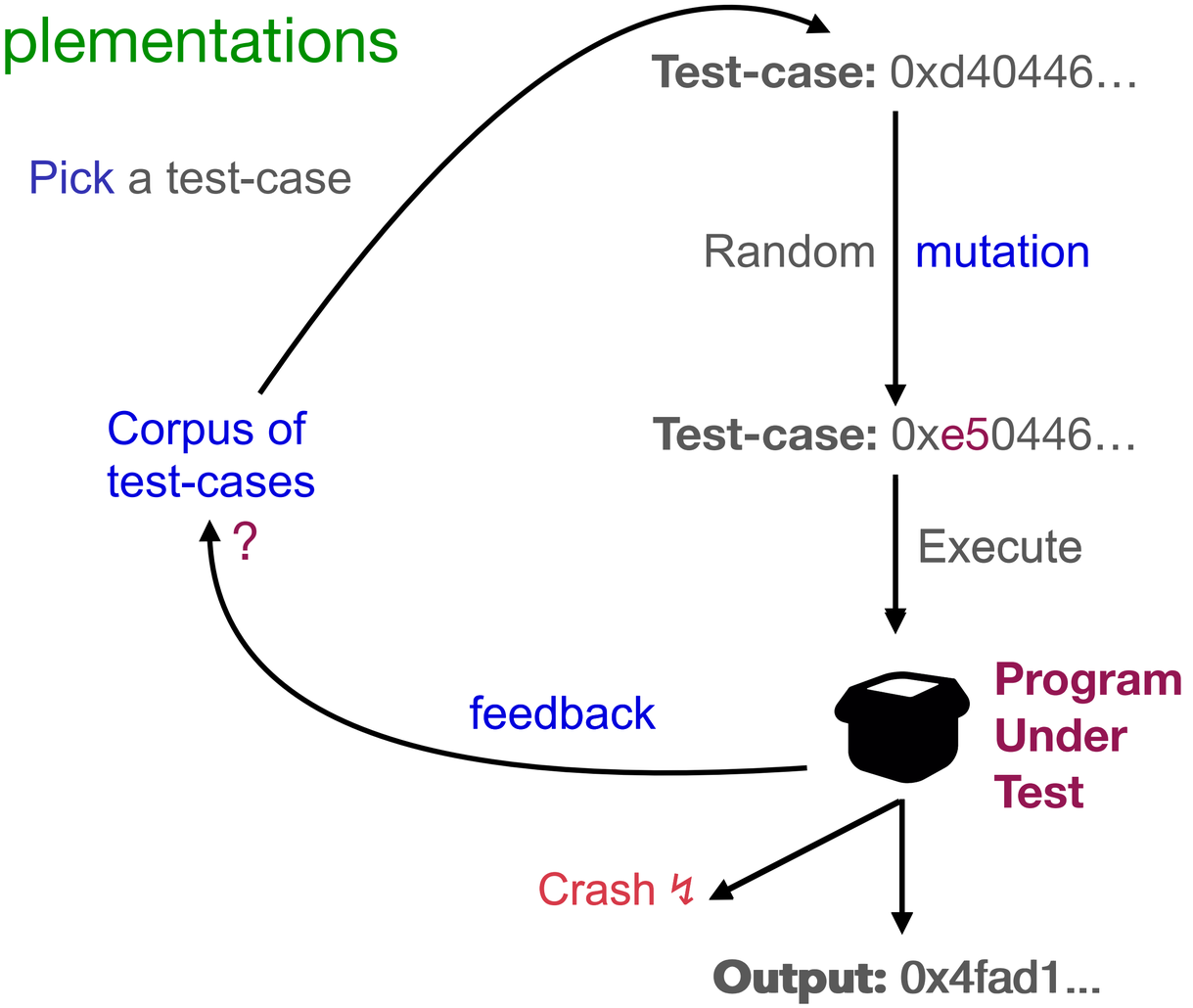
☞ Finds memory/crash vulnerabilities in implementations

E.g., buffer-overflow, use after free, RCE, etc.

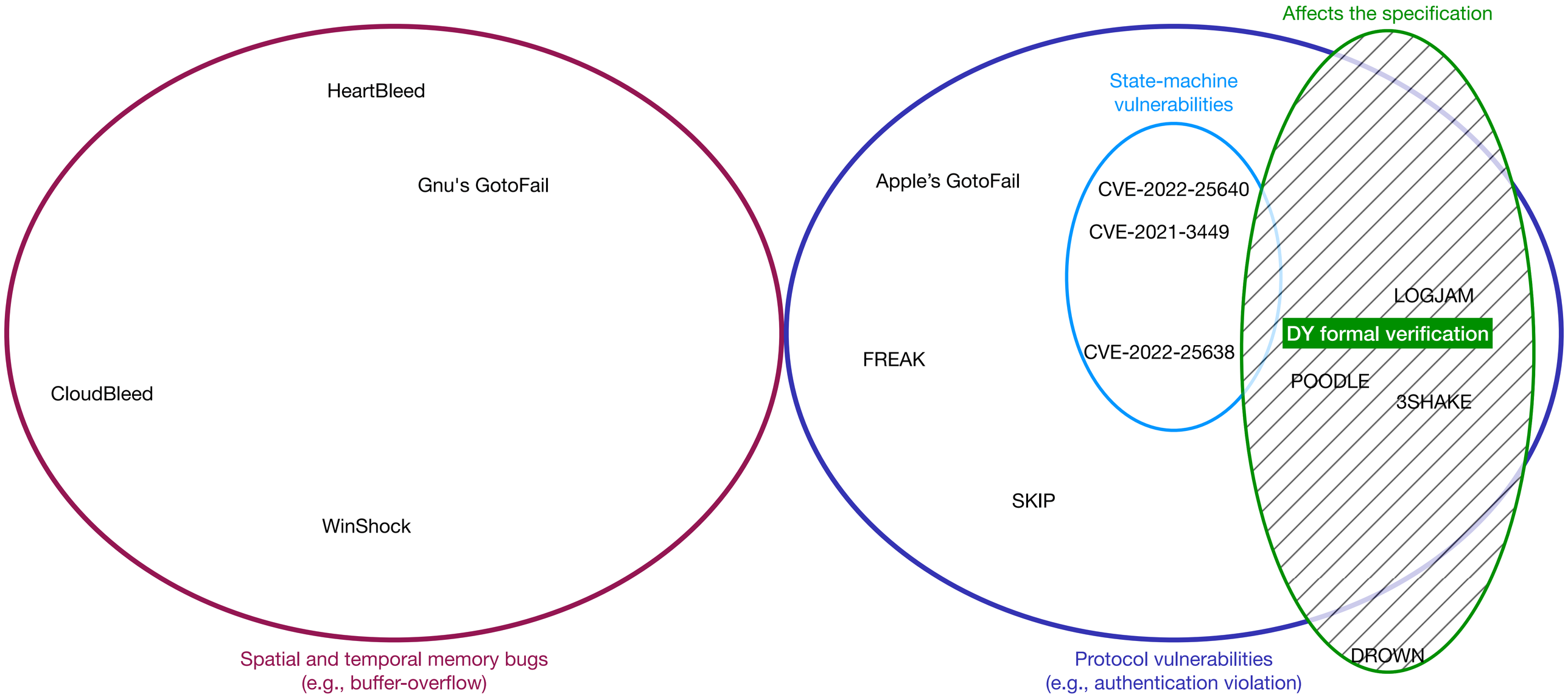
☞ Logical attack states are not reached
  ☞ miss some memory vulnerabilities

structural message modification:
e.g., negligible probability of finding a valid encryption through mutations
message flow modification

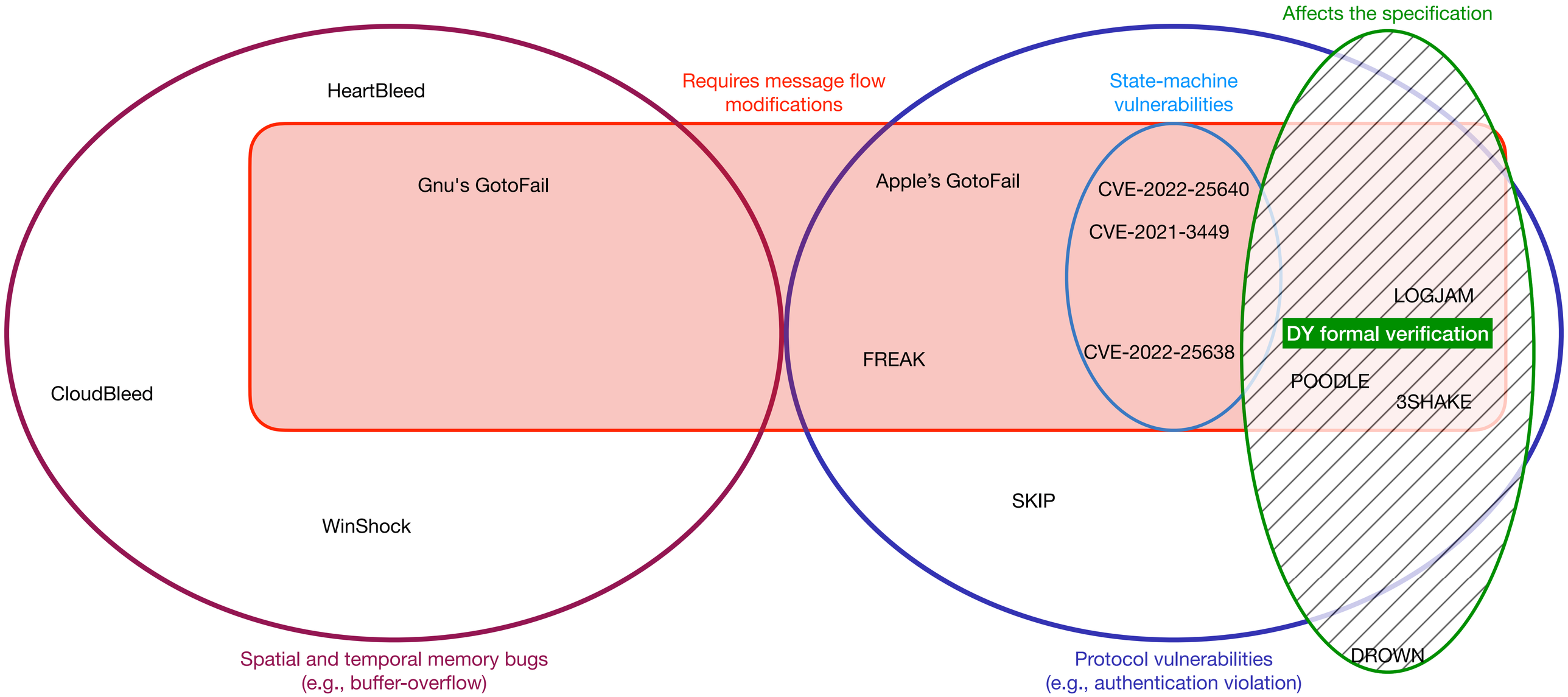
☞ Protocol vulnerabilities are not detected
e.g., authentication bypass (no crash)



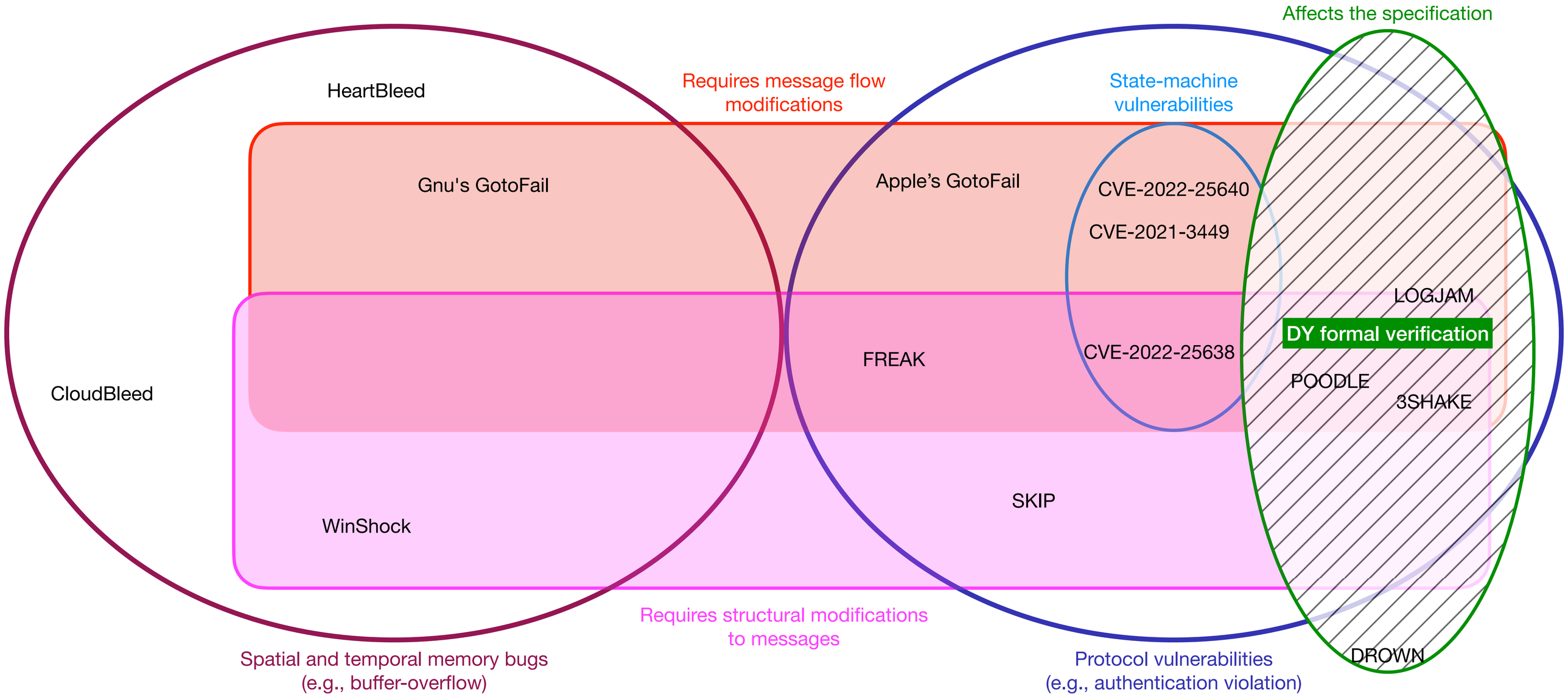
Retrospective of TLS Failures https:// 2014-2022



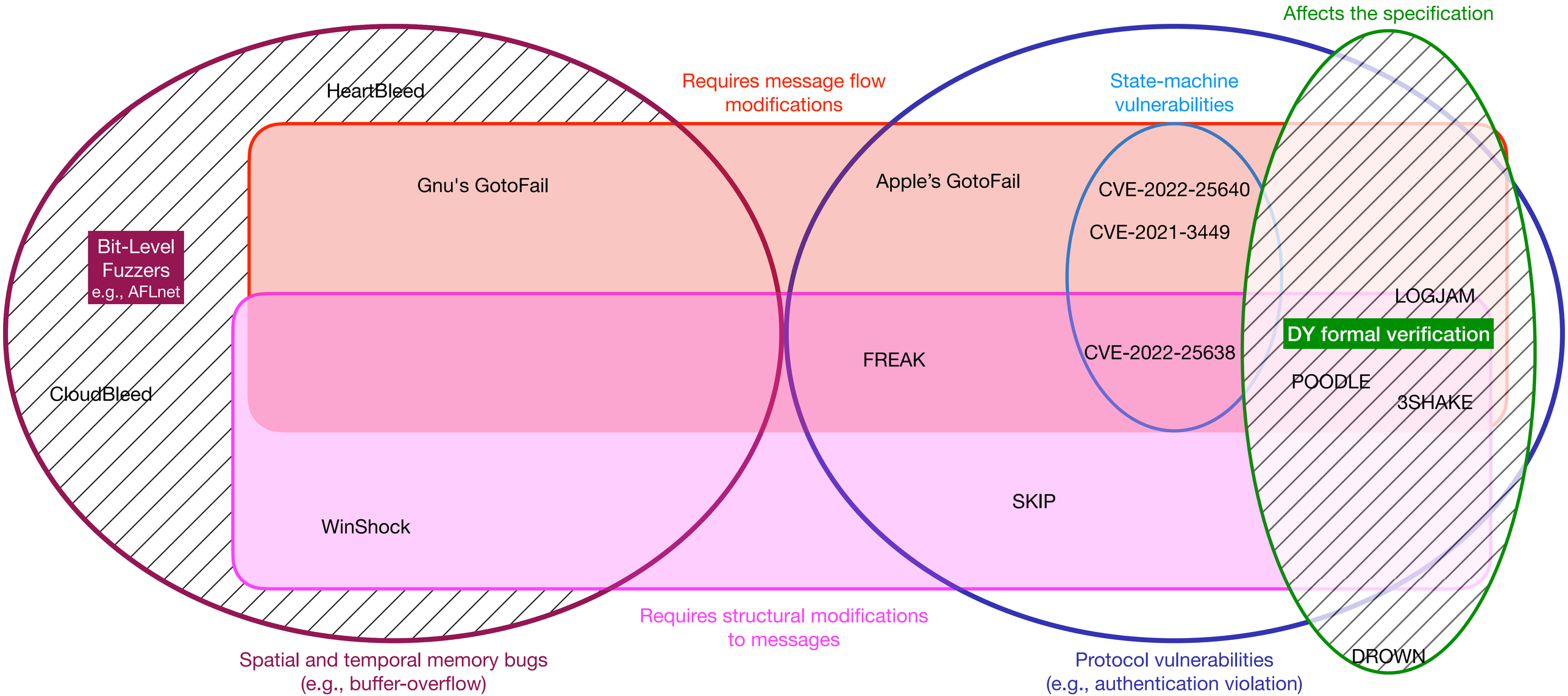
Retrospective of TLS Failures https:// 2014-2022



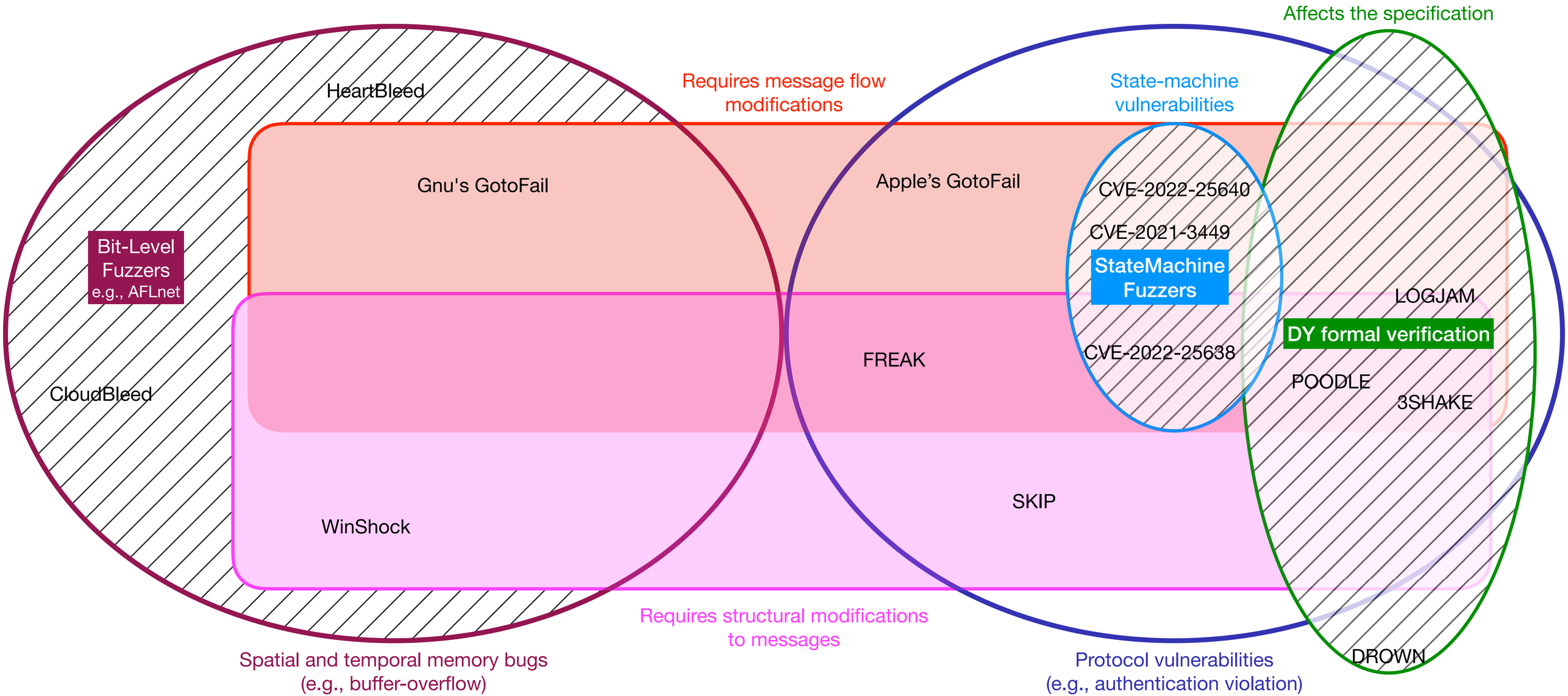
Retrospective of TLS Failures https:// 2014-2022



Retrospective of TLS Failures https:// 2014-2022



Retrospective of TLS Failures https:// 2014-2022



State-Machine-based Fuzzers

State-Machine-based Fuzzers

- State-machine fuzzers/learners where:
  - Input state = state-machine model: e.g., finite alphabet, automaton
  - Test-case = series of actions (from alphabet), e.g., ClientHello, ServerHello

State-Machine-based Fuzzers

- State-machine fuzzers/learners where:
  - Input state = state-machine model: e.g., finite alphabet, automaton
  - Test-case = series of actions (from alphabet), e.g., ClientHello, ServerHello

+ Captures protocol vulnerabilities corresponding to state-machine violations in implementations, e.g., [22-25640] SkipVerify (auth. bypass with CertInfo but wo/ Cert)

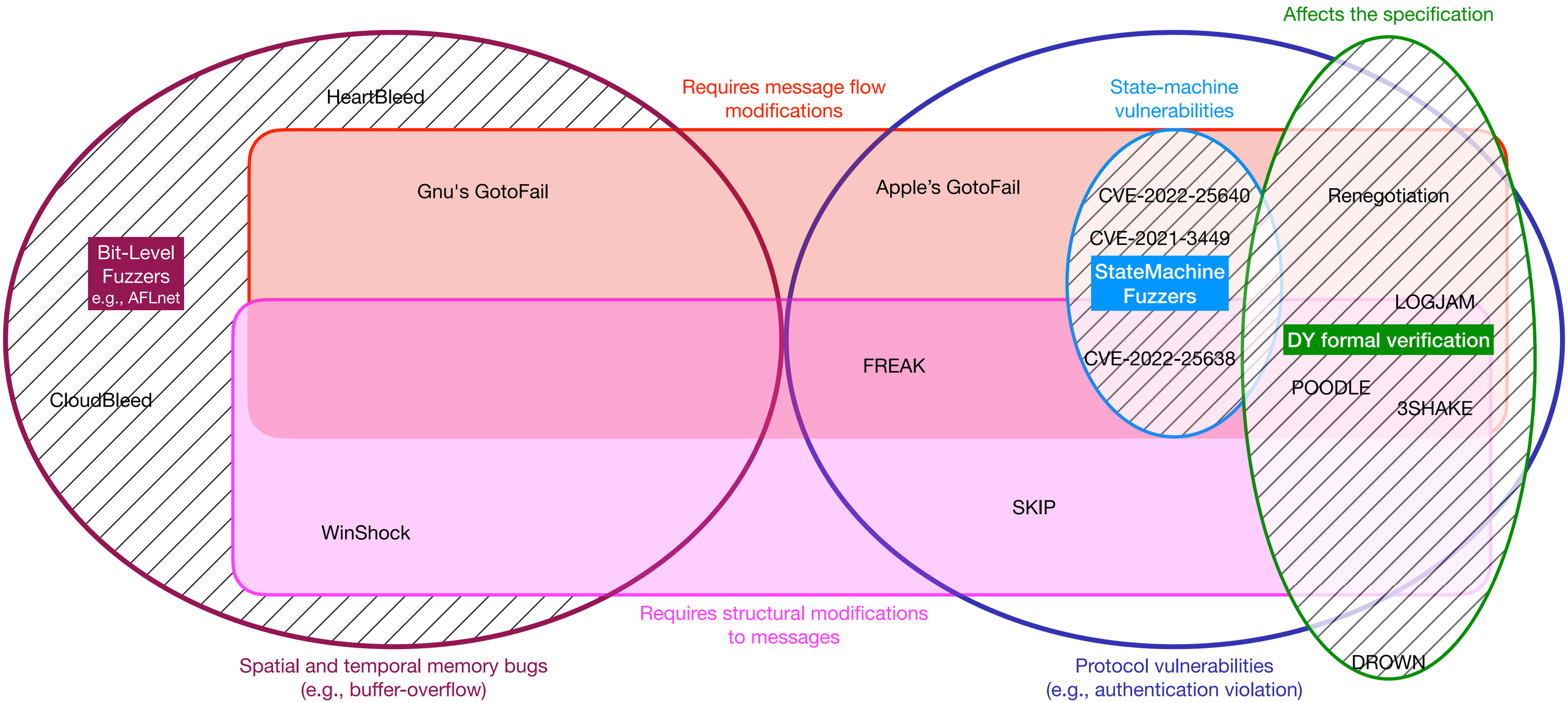
State-Machine-based Fuzzers

- State-machine fuzzers/learners where:
  - Input state = state-machine model: e.g., finite alphabet, automaton
  - Test-case = series of actions (from alphabet), e.g., ClientHello, ServerHello

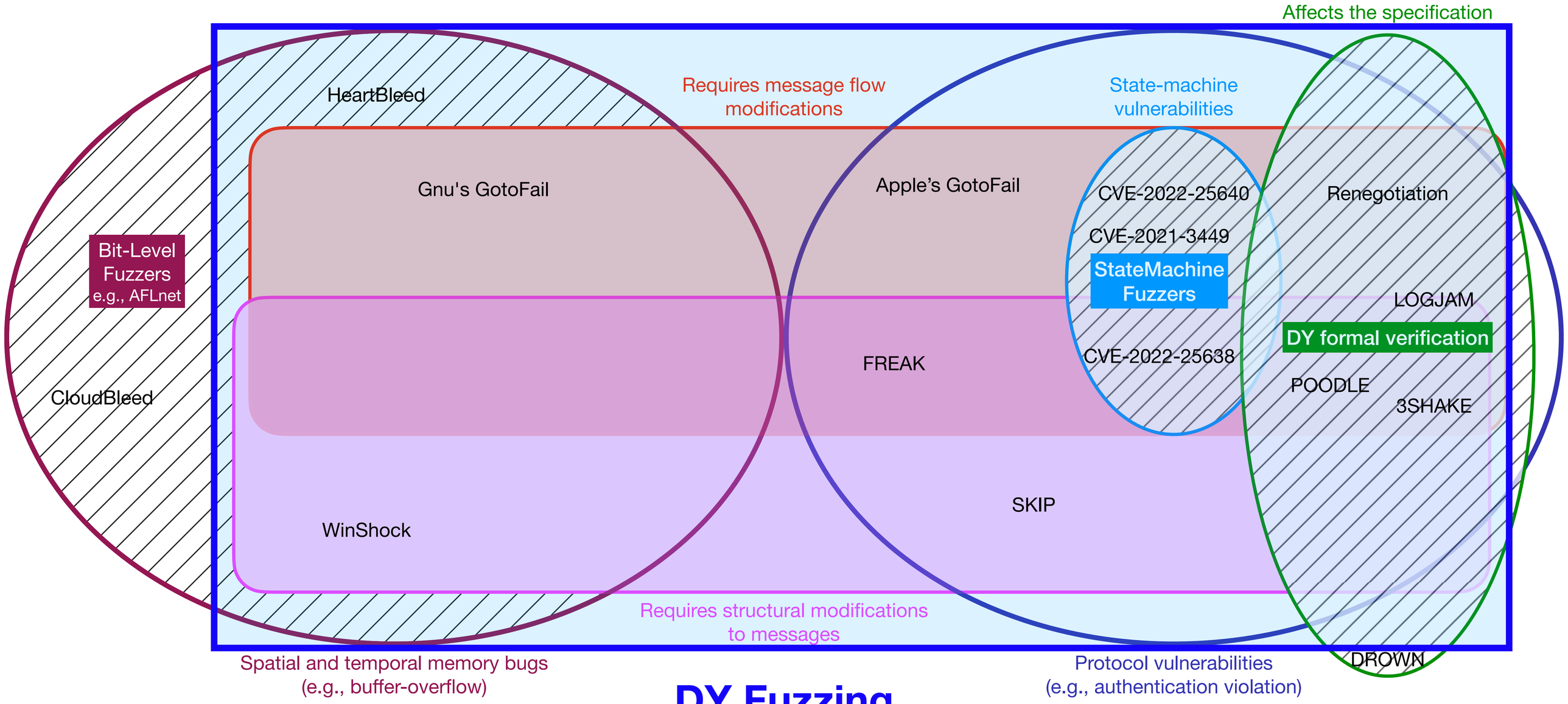
+ Captures protocol vulnerabilities corresponding to state-machine violations in implementations, e.g., [22-25640] SkipVerify (auth. bypass with CertInfo but wo/ Cert)

- Reachability: no (structural) message modifications (except finite built-in modifs)
  ☞ does not capture the class of logical attacks
- Detection: manual+incomplete vulnerability detections (memory/protocol vulns)

Retrospective of TLS Failures https:// 2014-2022

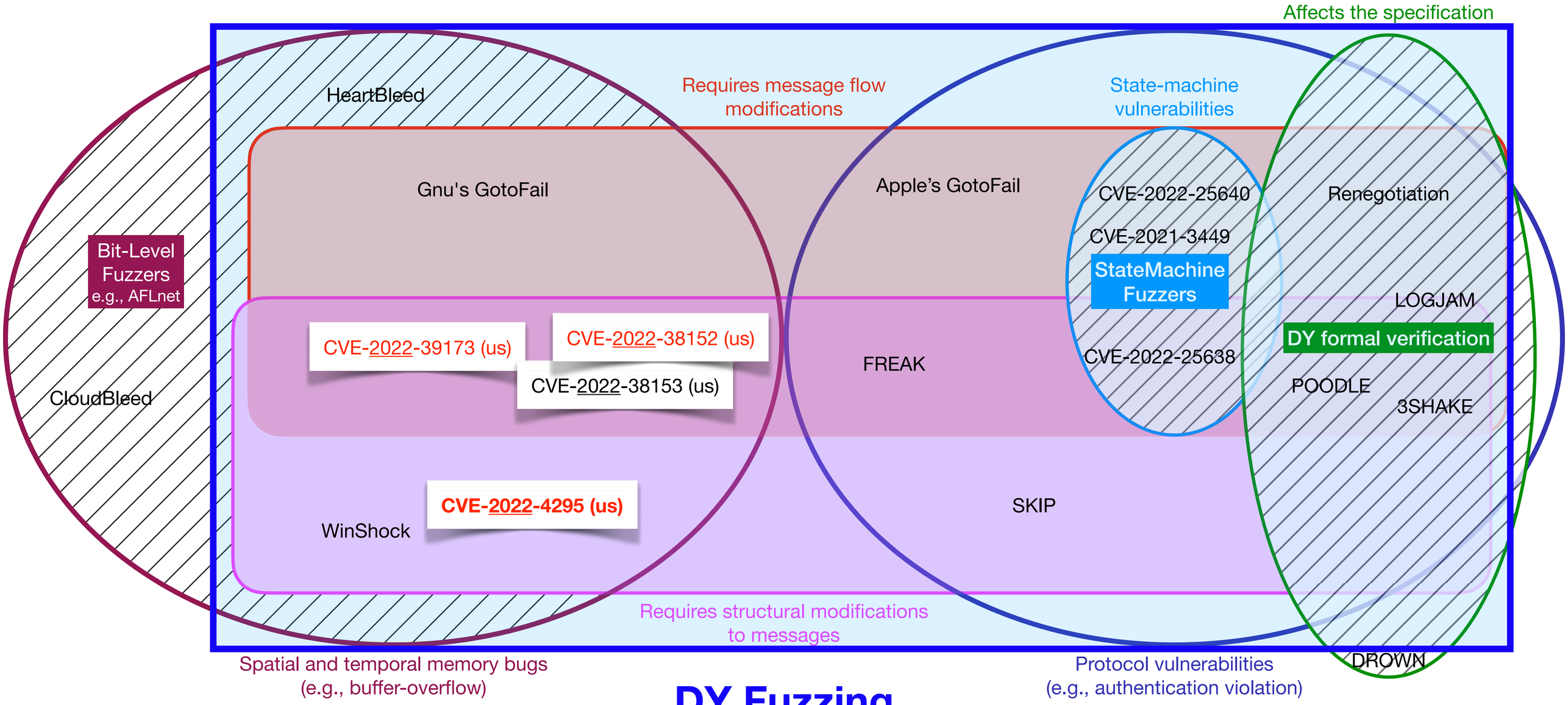


Retrospective of TLS Failures https:// 2014-2022



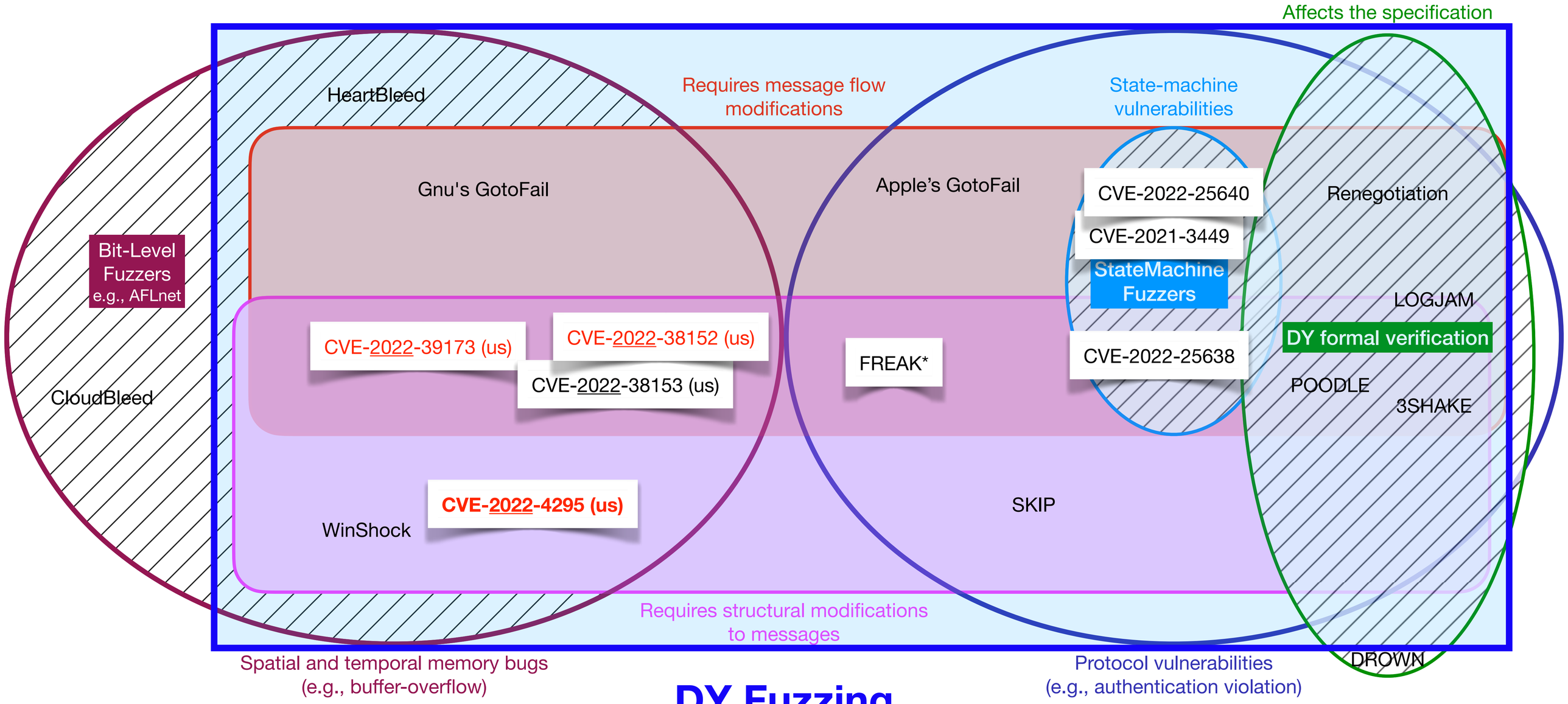
DY Fuzzing

Retrospective of TLS Failures https:// 2014-2022



DY Fuzzing

Retrospective of TLS Failures https:// 2014-2022



DY Fuzzing

DY Fuzzing Design

DY Fuzzing: Big Picture

DY Fuzzer = DY attacker in a fuzzing loop

DY Fuzzing: Big Picture

DY Fuzzer = DY attacker in a fuzzing loop

- We build on "messages as formal terms" and assume a term algebra

DY Fuzzing: Big Picture

DY Fuzzer = DY attacker in a fuzzing loop

- We build on "messages as formal terms" and assume a term algebra

- Test cases = symbolic traces expressing DY attacker's actions

```
tr := out(r, w).tr   : r is a role (client/server) and w is a variable (attacker knows)
    | in(r, R).tr    : R is a term in the term algebra (computed by attacker)
    | 0
```

DY Fuzzing: Big Picture

DY Fuzzer = DY attacker in a fuzzing loop

- We build on "messages as formal terms" and assume a term algebra

- Test cases = symbolic traces expressing DY attacker's actions

  tr := out(r, w).tr : r is a role (client/server) and w is a variable (attacker knows)
  | in(r, R).tr : R is a term in the term algebra (computed by attacker)
  | 0

Example: tr_a=out(cl,w_1).in(serv,w_1).out(serv,w_2).in(cl, sign(extract(w_2), sk_a)).0

DY Fuzzing: Big Picture

DY Fuzzer = MITM DY attacker

```
tr := out(r, w).tr
    |  in(r, R).tr
    | 0
```

DY Fuzzing: Big Picture

DY Fuzzer = MITM DY attacker

Symbolic traces (tr) are « concretized » with the PUT (or any ref. implem.)

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

DY Fuzzing: Big Picture

DY Fuzzer = MITM DY attacker

Symbolic traces (tr) are « concretized » with the PUT (or any ref. implem.)

1. out(r, w) ☞ call PUT role function to read bitstring b_w from output buffer of r

```
tr := out(r, w).tr
    |  in(r, R).tr
    | 0
```

DY Fuzzing: Big Picture

DY Fuzzer = MITM DY attacker

Symbolic traces (tr) are « concretized » with the PUT (or any ref. implem.)

1. out(r, w) ☞ call PUT role function to read bitstring b_w from output buffer of r
2. in(r, R) ☞
   a. call ref/PUT crypto functions to evaluate R into a bitstring b_R

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

DY Fuzzing: Big Picture

DY Fuzzer = MITM DY attacker

Symbolic traces (tr) are « concretized » with the PUT (or any ref. implem.)

1. out(r, w) ☞ call PUT role function to read bitstring b_w from output buffer of r
2. in(r, R) ☞
   a. call ref/PUT crypto functions to evaluate R into a bitstring b_R

      E.g., $eval(sign(R',sk)) = RSA_{PUT}(eval(R'),b_{sk})$

      $eval(w) = b_w$

      b_{sk} is obtained by calling $genKey_{PUT}()$

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

DY Fuzzing: Big Picture

DY Fuzzer = MITM DY attacker

Symbolic traces (tr) are « concretized » with the PUT (or any ref. implem.)

1. out(r, w) ☞ call PUT role function to read bitstring b_w from output buffer of r
2. in(r, R) ☞
   a. call ref/PUT crypto functions to evaluate R into a bitstring b_R
      E.g., $eval(sign(R',sk)) = RSA_{PUT}(eval(R'),b_{sk})$
      $eval(w) = b_w$
      b_{sk} is obtained by calling $genKey_{PUT}()$
   b. call PUT role function to write b_R onto input buffer of r + make r progress

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

DY Fuzzing: Big Picture

DY Fuzzer = MITM DY attacker

Symbolic traces (tr) are « concretized » with the PUT (or any ref. implem.)

1. out(r, w) ☞ call PUT role function to read bitstring b_w from output buffer of r
2. in(r, R) ☞
   a. call ref/PUT crypto functions to evaluate R into a bitstring b_R
      E.g., $eval(sign(R',sk)) = RSA_{PUT}(eval(R'),b_{sk})$
      $eval(w) = b_w$
      b_{sk} is obtained by calling $genKey_{PUT}()$
   b. call PUT role function to write b_R onto input buffer of r + make r progress

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

Executor (1 + 2.b): require a lightweight instrumentation of the PUT
Mapper (2.a): requires a per-protocol « executable term-algebra »

DY Fuzzing: Big Picture

DY Fuzzer = MITM DY attacker

Symbolic traces (tr) are « concretized » with the PUT (or any ref. implem.)

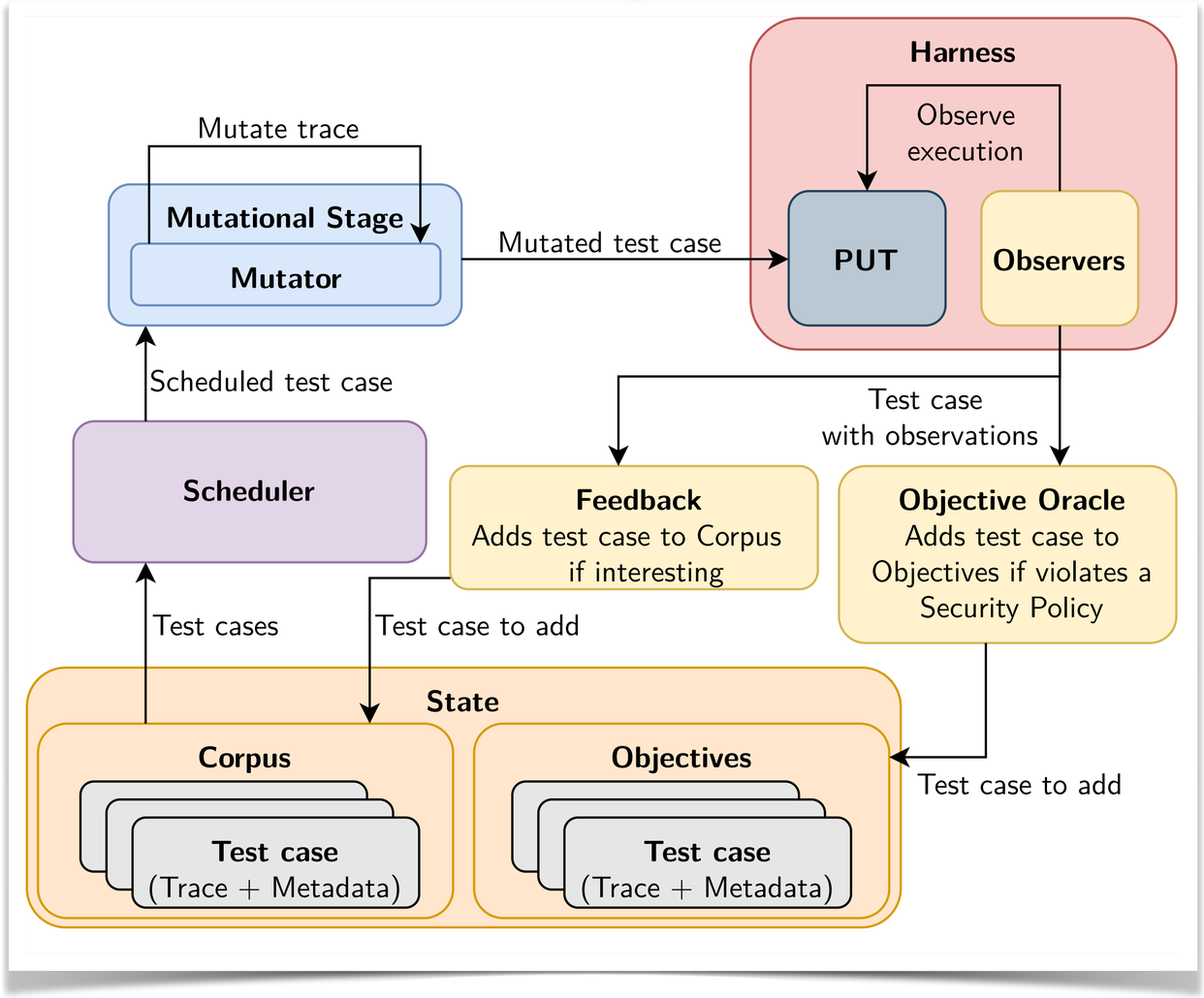
1. out(r, w) ☞ call PUT role function to read bitstring b_w from output buffer of r
2. in(r, R) ☞
   a. call ref/PUT crypto functions to evaluate R into a bitstring b_R
      E.g., $eval(sign(R',sk)) = RSA_{PUT}(eval(R'),b_{sk})$
      $eval(w) = b_w$
      b_{sk} is obtained by calling $genKey_{PUT}()$
   b. call PUT role function to write b_R onto input buffer of r + make r progress

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

Executor (1 + 2.b): require a lightweight instrumentation of the PUT
Mapper (2.a): requires a per-protocol « executable term-algebra »

☞ Do **not** require a protocol DY model but only a DY attacker model (*i.e.*, term algebra)

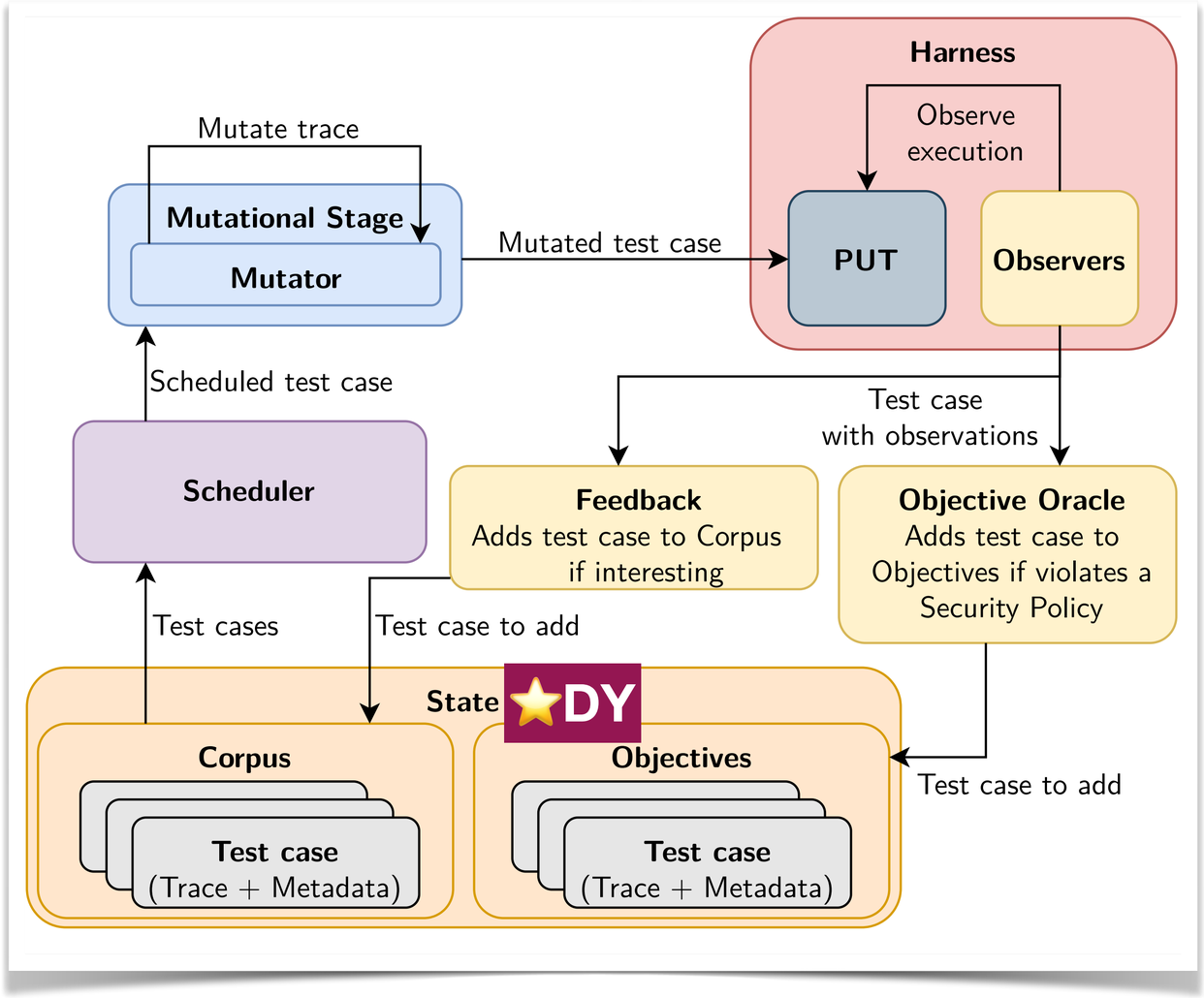
DY Fuzzer components



LibAFL components (we build on)

DY Fuzzer components

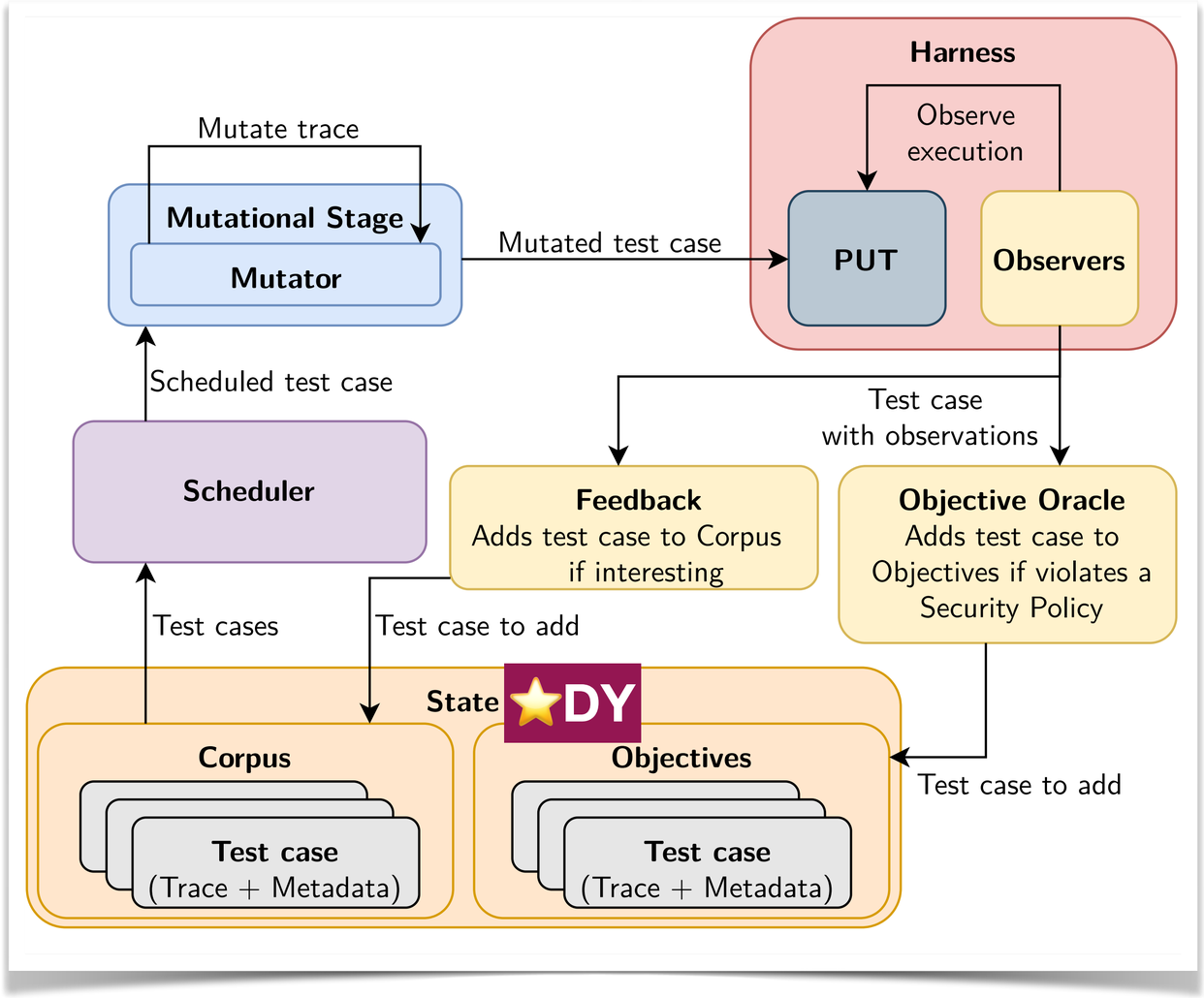
- State⭐: test-cases = DY traces,
  seeds corpus = happy flows



LibAFL components (we build on)

DY Fuzzer components

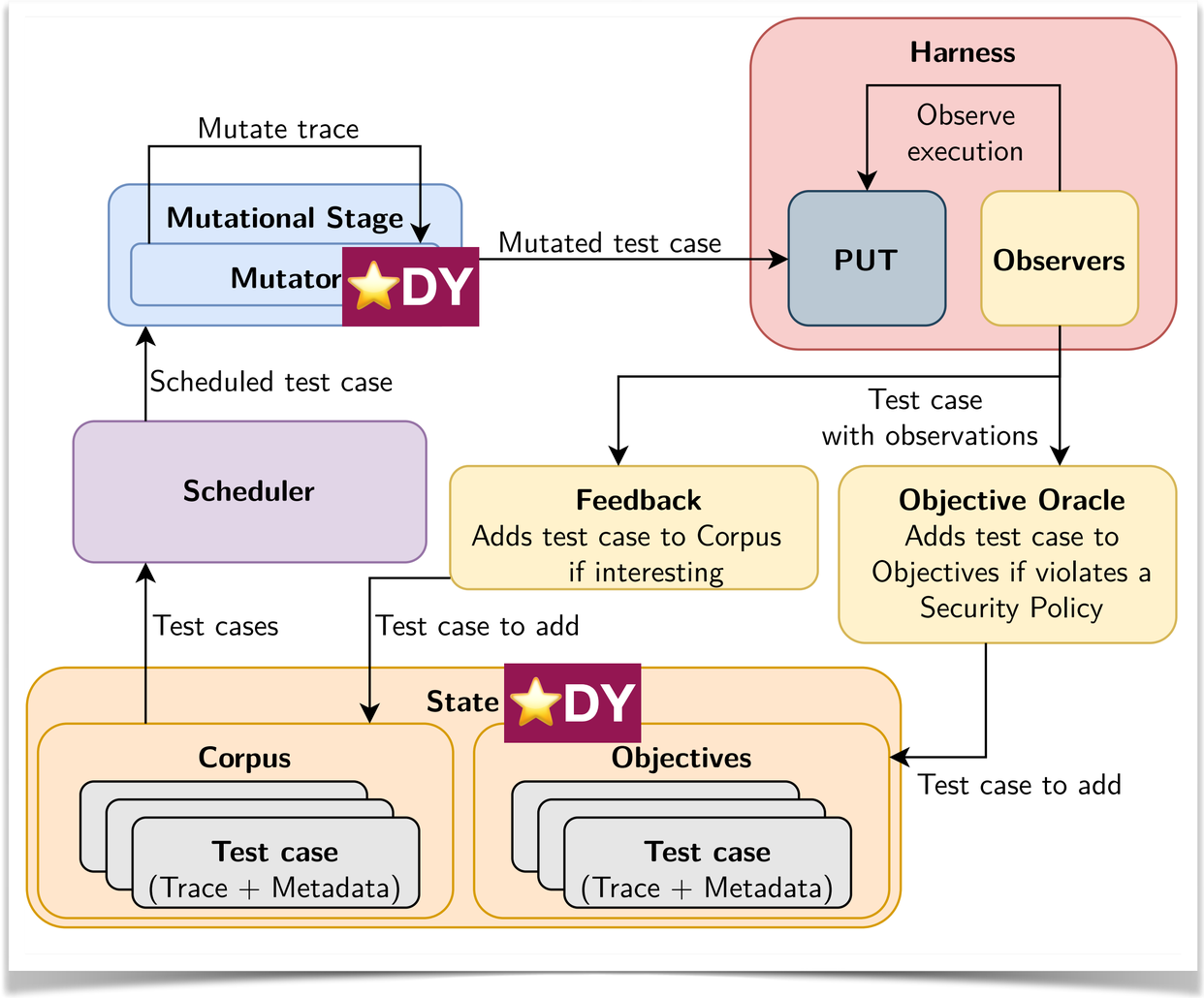
- State⭐: test-cases = DY traces,
  seeds corpus = happy flows
- Scheduler: FIFO



LibAFL components (we build on)

DY Fuzzer components

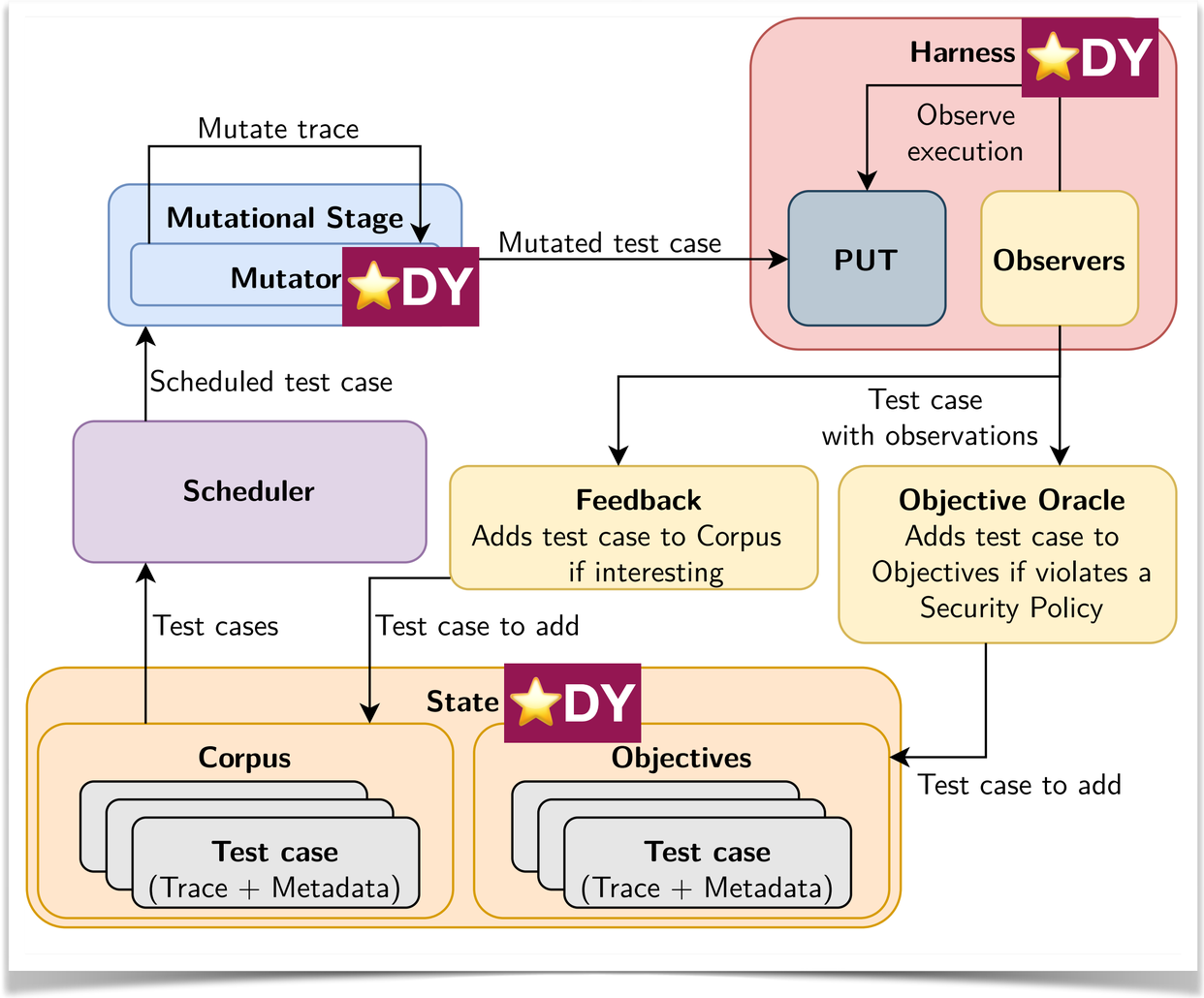
- State⭐: test-cases = DY traces,
  seeds corpus = happy flows
- Scheduler: FIFO
- Mutator⭐: custom trace mutations



LibAFL components (we build on)

DY Fuzzer components

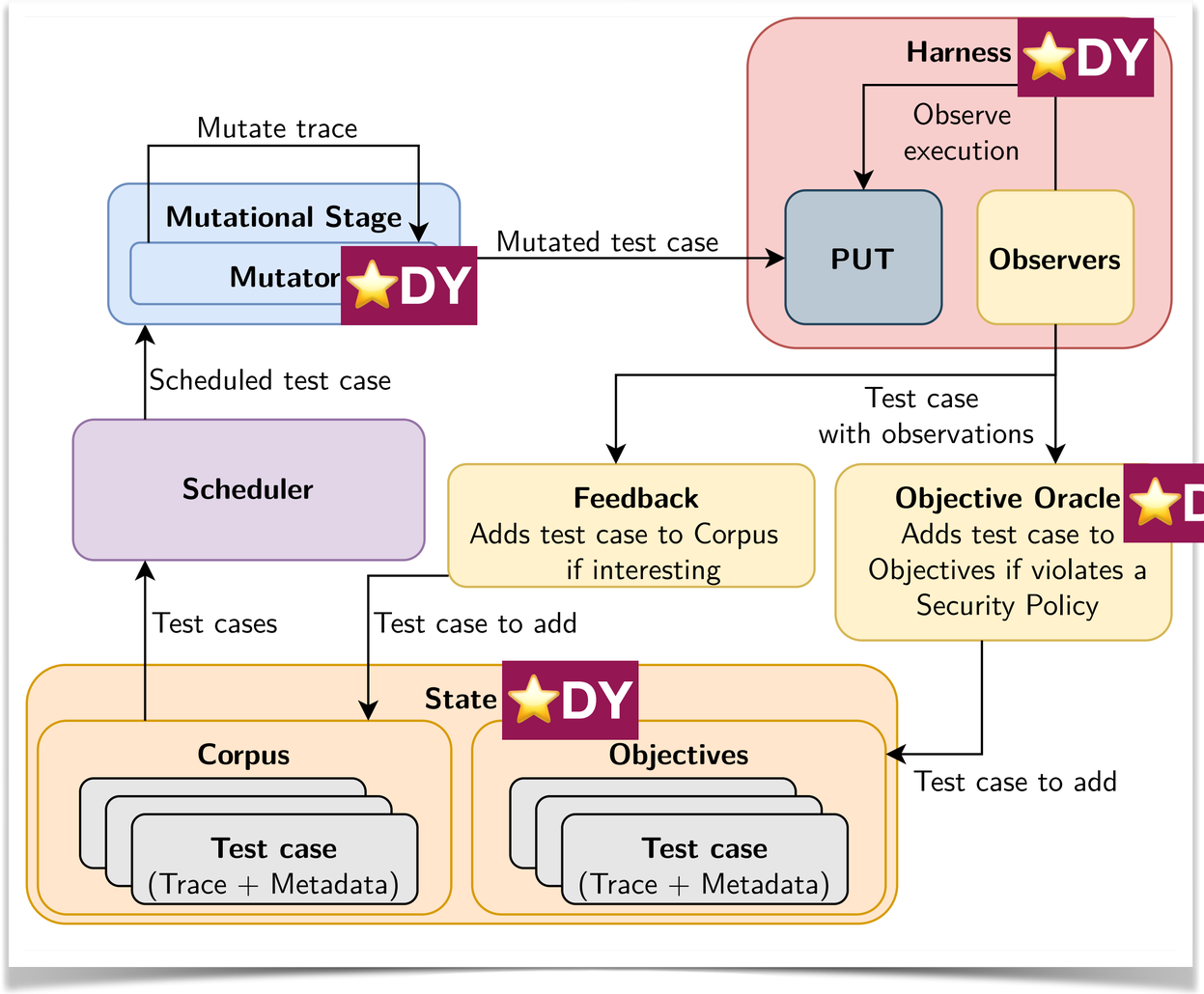
- State⭐: test-cases = DY traces,
  seeds corpus = happy flows
- Scheduler: FIFO
- Mutator⭐: custom trace mutations
- Harness⭐: Mapper + Executor + Claims



LibAFL components (we build on)

DY Fuzzer components

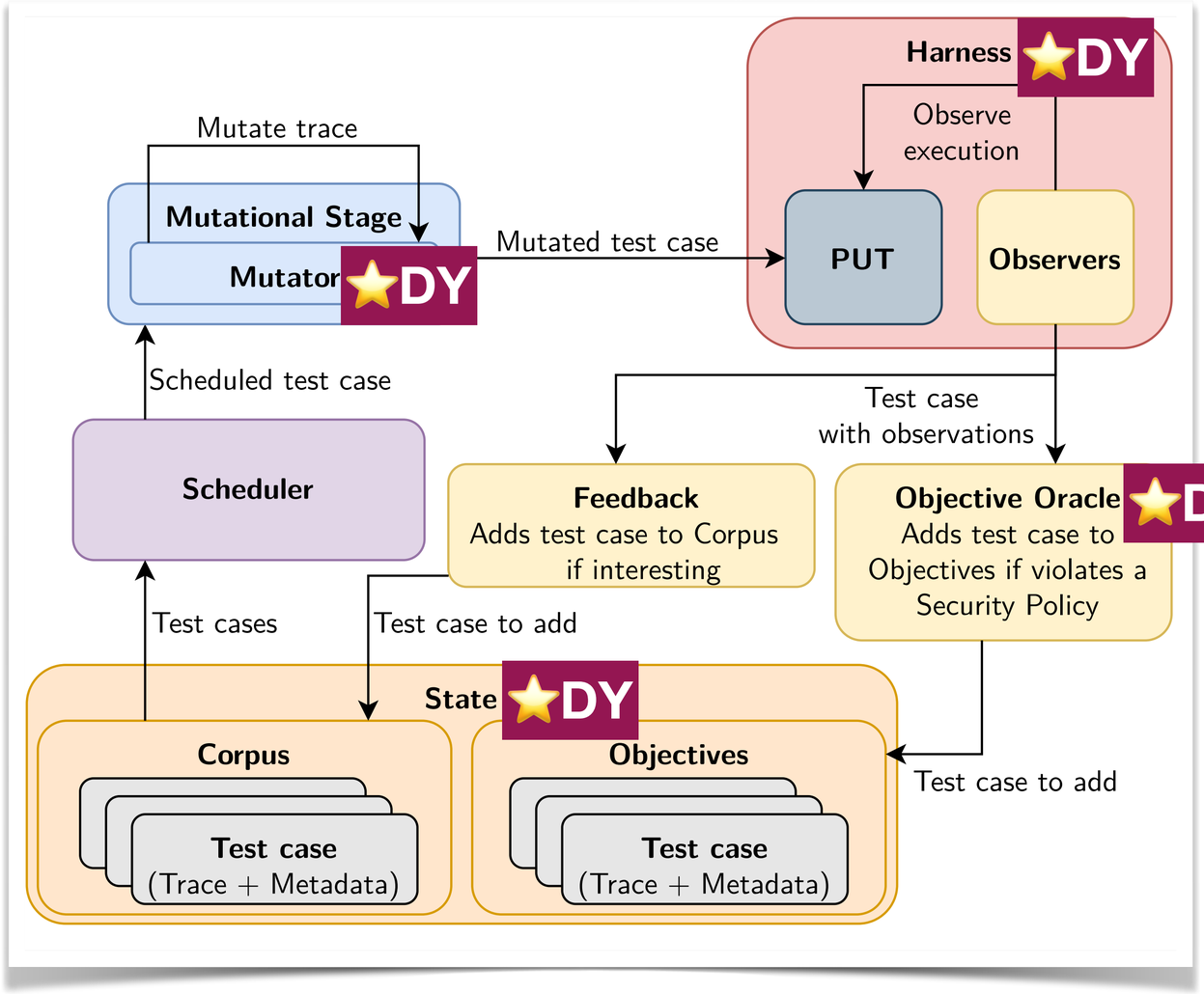
- State⭐: test-cases = DY traces,
  seeds corpus = happy flows
- Scheduler: FIFO
- Mutator⭐: custom trace mutations
- Harness⭐: Mapper + Executor + Claims
- Obj. Oracle⭐: DY security properties⭐
  (e.g., agreement) + ASAN (memory vulns.)



LibAFL components (we build on)

DY Fuzzer components

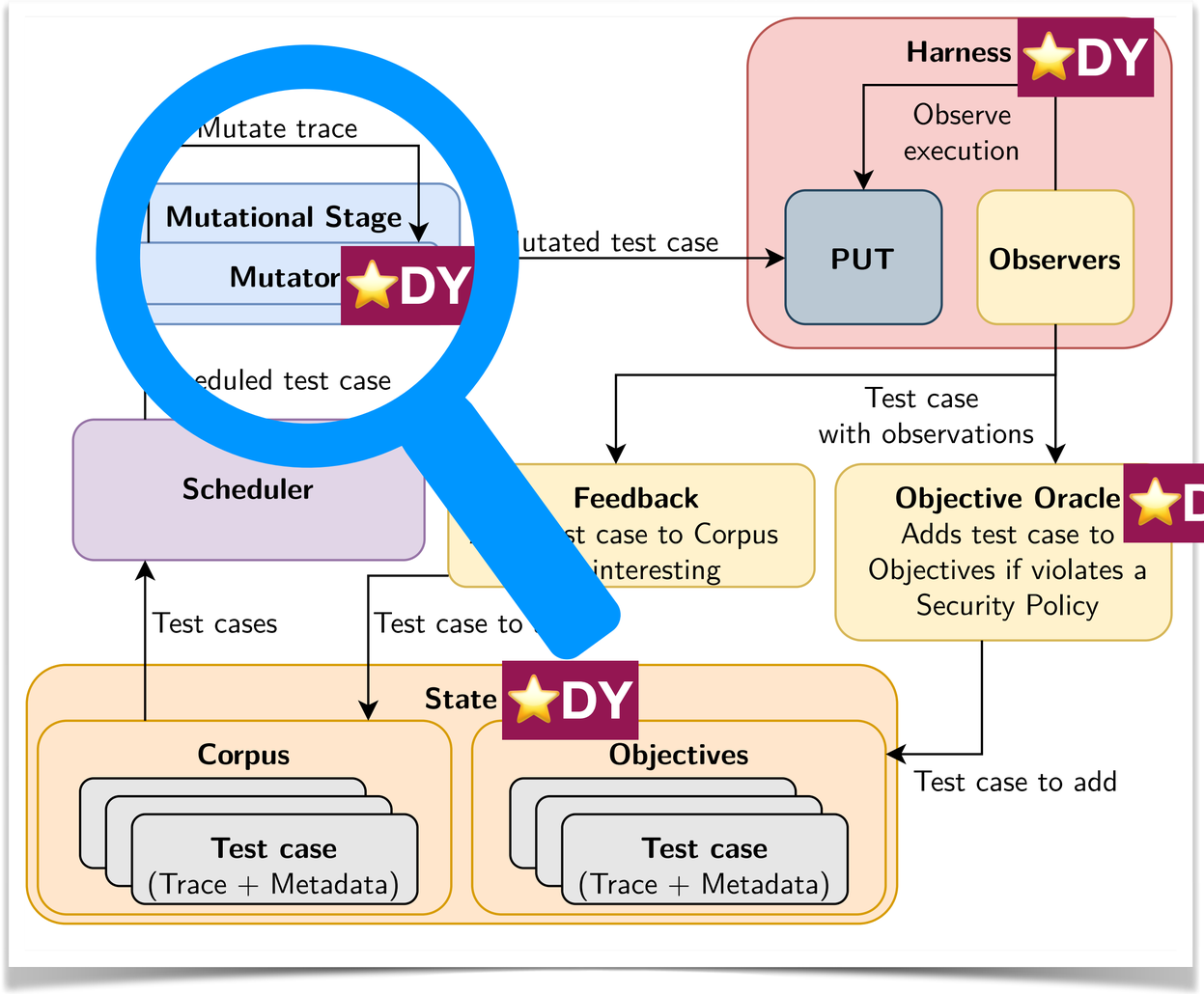
- State⭐: test-cases = DY traces,
  seeds corpus = happy flows
- Scheduler: FIFO
- Mutator⭐: custom trace mutations
- Harness⭐: Mapper + Executor + Claims
- Obj. Oracle⭐: DY security properties⭐
  (e.g., agreement) + ASAN (memory vulns.)
- Feedback: PUT code-coverage



LibAFL components (we build on)

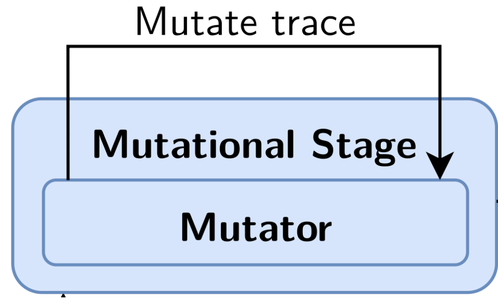
DY Fuzzer components

- State⭐: test-cases = DY traces,
  seeds corpus = happy flows
- Scheduler: FIFO
- Mutator⭐: custom trace mutations
- Harness⭐: Mapper + Executor + Claims
- Obj. Oracle⭐: DY security properties⭐
  (e.g., agreement) + ASAN (memory vulns.)
- Feedback: PUT code-coverage



LibAFL components (we build on)

DY mutations

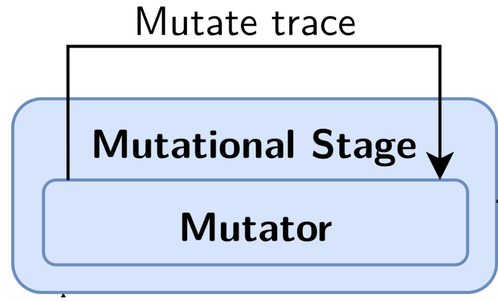



```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

DY mutations

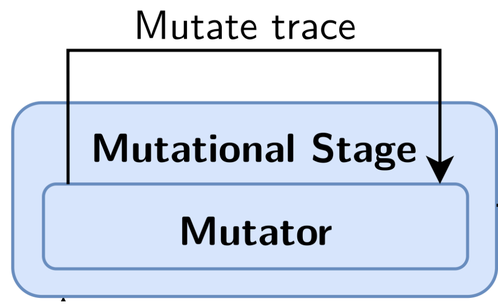
Action-level Mutations

- Skip: remove random action (in/out)




```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

DY mutations



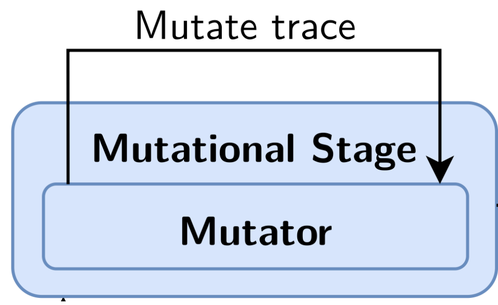
Action-level Mutations

- Skip: remove random action (in/out)
- Repeat: randomly copy and insert an action

☞ Already enough to capture some state-machine vulns such as auth. bypass

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

DY mutations



Action-level Mutations

- Skip: remove random action (in/out)
- Repeat: randomly copy and insert an action

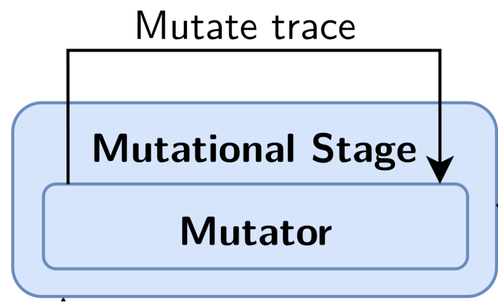
☞ Already enough to capture some state-machine vulns such as auth. bypass

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

Term-level Mutations⭐

- Swap: Swap two (sub-)terms in the trace

DY mutations



Action-level Mutations

- Skip: remove random action (in/out)
- Repeat: randomly copy and insert an action

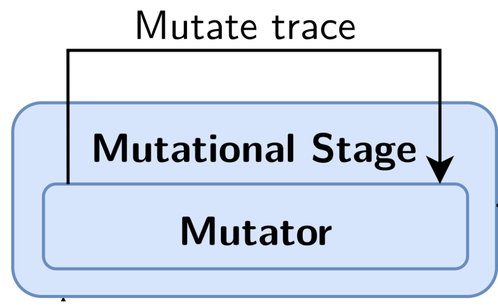
☞ Already enough to capture some state-machine vulns such as auth. bypass

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

Term-level Mutations⭐

- Swap: Swap two (sub-)terms in the trace
- Generate: Replace a term by a random one

DY mutations



Action-level Mutations

- Skip: remove random action (in/out)
- Repeat: randomly copy and insert an action

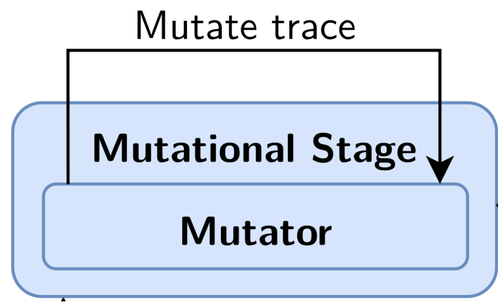
☞ Already enough to capture some state-machine vulns such as auth. bypass

```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

Term-level Mutations⭐

- Swap: Swap two (sub-)terms in the trace
- Generate: Replace a term by a random one
- Replace-Match: Swap two function symbols in the trace (e.g., SHA2 <-> SHA3)

DY mutations



Action-level Mutations

- Skip: remove random action (in/out)
- Repeat: randomly copy and insert an action

☞ Already enough to capture some state-machine vulns such as auth. bypass

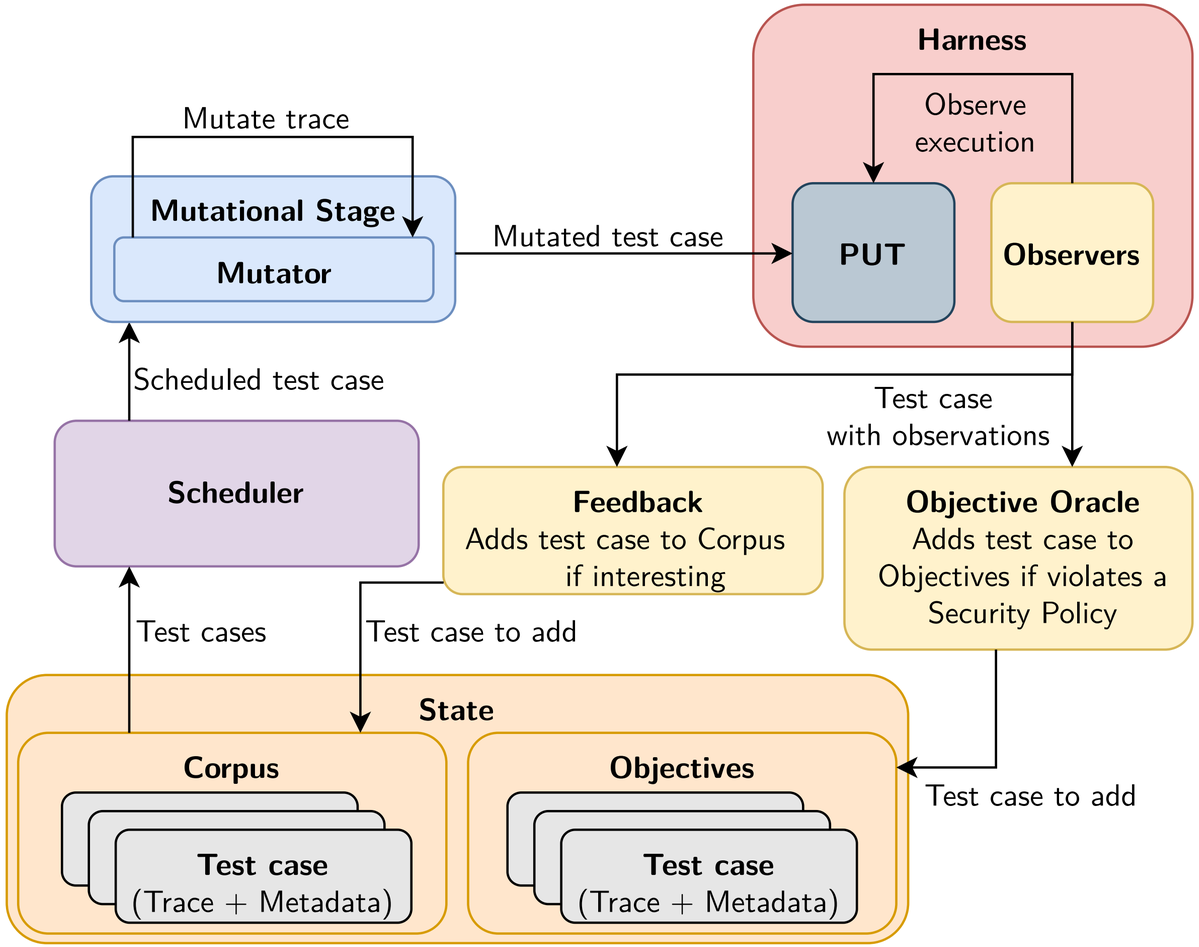
```
tr := out(r, w).tr
    | in(r, R).tr
    | 0
```

Term-level Mutations⭐

- Swap: Swap two (sub-)terms in the trace
- Generate: Replace a term by a random one
- Replace-Match: Swap two function symbols in the trace (e.g., SHA2 <-> SHA3)
- Replace-Reuse: Replace a (sub-)term by another (sub-)term in the trace
- Replace-and-Lift: Replace a (sub-)term by one of its sub-terms

☞ Mutations are conditioned: well-typed (avoid systematic failures) + size-bounds

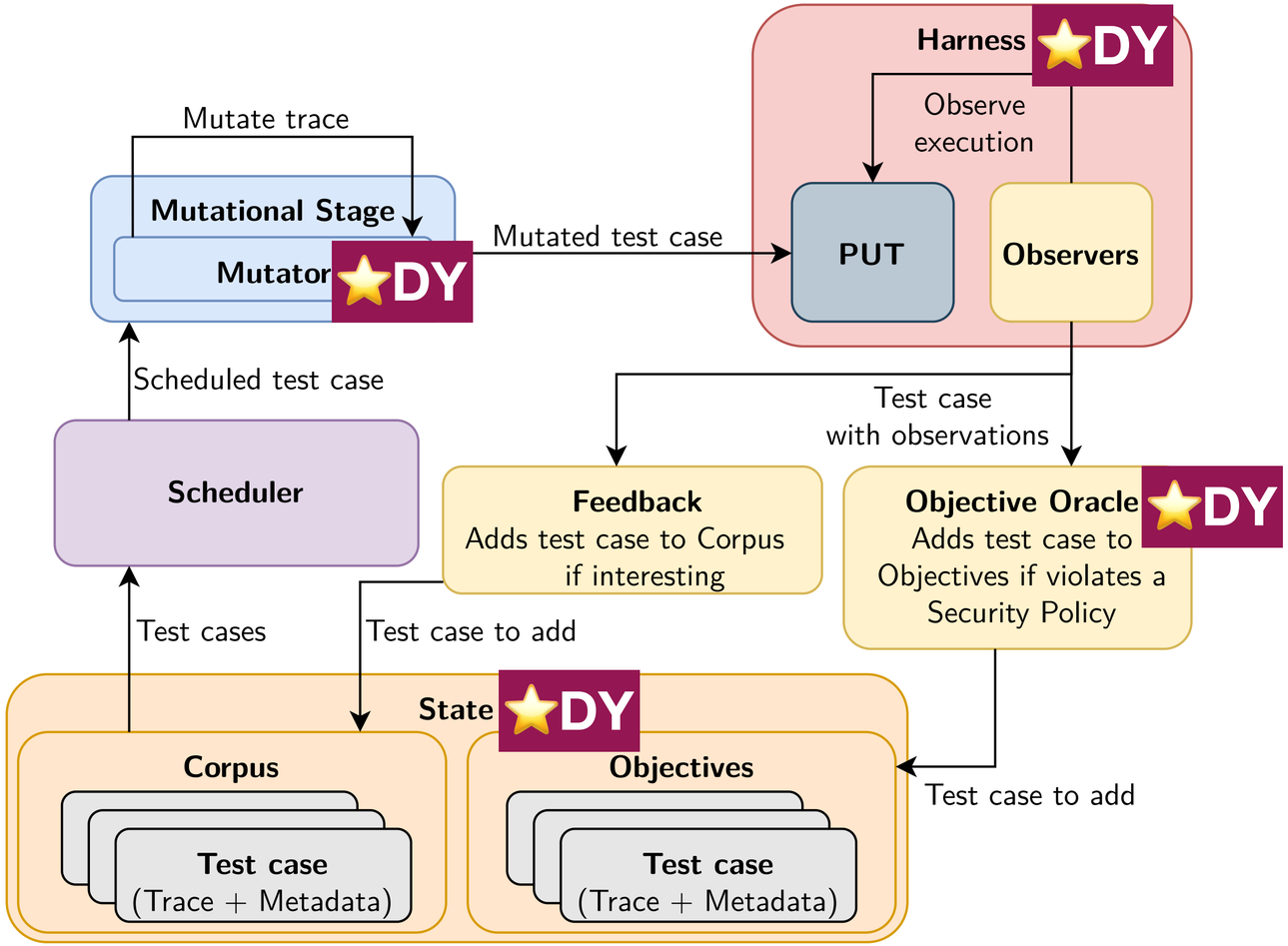
DY Fuzzer components



LibAFL components (we build on)

DY Fuzzer components

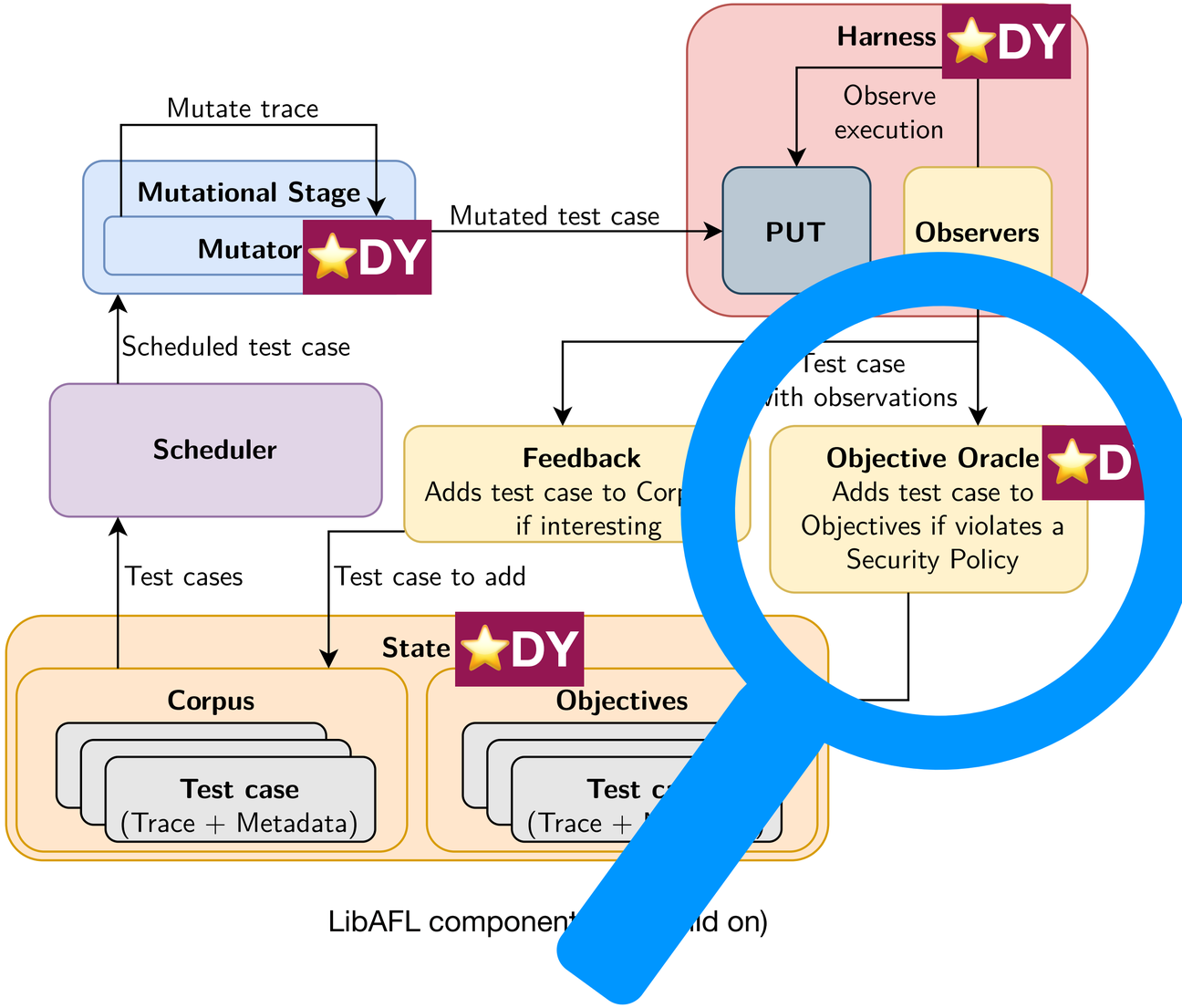
- State⭐: test-cases = DY traces,
  seeds corpus = happy flows
- Scheduler: FIFO
- Mutator⭐: custom trace mutations
- Harness⭐: Mapper + Executor + Claims
- Obj. Oracle⭐: DY security properties⭐
  (e.g., agreement) + ASAN (memory vulns.)
- Feedback: PUT code-coverage



LibAFL components (we build on)

DY Fuzzer components

- State⭐: test-cases = DY traces,
  seeds corpus = happy flows
- Scheduler: FIFO
- Mutator⭐: custom trace mutations
- Harness⭐: Mapper + Executor + Claims
- Obj. Oracle⭐: DY security properties⭐
  (e.g., agreement) + ASAN (memory vulns.)
- Feedback: PUT code-coverage



DY Objective Oracle

Objective Oracle
Adds test case to Objectives if violates a Security Policy

DY Objective Oracle

Objective Oracle
Adds test case to Objectives if violates a Security Policy

Memory-related objective oracle

- Classical with bit-level fuzzing: code instrumentation with AddressSanitizer (*ASan*)

DY Objective Oracle

Objective Oracle
Adds test case to Objectives if violates a Security Policy

Memory-related objective oracle

- Classical with bit-level fuzzing: code instrumentation with AddressSanitizer (*ASan*)

+

DY Security properties ⭐

- Introduce claims triggered by roles executing the PUT (part of Harness/Executor)
  E.g., agreement claims: Agr(client, pk, m)@i client believes to have agreed with server with pk on m @ i^{th} action

DY Objective Oracle

Objective Oracle
Adds test case to Objectives if violates a Security Policy

Memory-related objective oracle

- Classical with bit-level fuzzing: code instrumentation with AddressSanitizer (*ASan*)

+

DY Security properties ⭐

- Introduce claims triggered by roles executing the PUT (part of Harness/Executor)
  E.g., agreement claims: Agr(client, pk, m)@i client believes to have agreed with server with pk on m @ i^{th} action
- Classical in DY models: security properties expressed as 1^{st}-order formula
  E.g., agreement property ∀pk,m: Agr(client, pk, m)@i ⇒ Run(server, pk, m)@j ∧ j<i

DY Objective Oracle

Objective Oracle
Adds test case to Objectives if violates a Security Policy

Memory-related objective oracle

- Classical with bit-level fuzzing: code instrumentation with AddressSanitizer (*ASan*)

+

DY Security properties ⭐

- Introduce claims triggered by roles executing the PUT (part of Harness/Executor)
  E.g., agreement claims: Agr(client, pk, m)@i client believes to have agreed with server with pk on m @ i^{th} action
- Classical in DY models: security properties expressed as 1^{st}-order formula
  E.g., agreement property ∀pk,m: Agr(client, pk, m)@i ⇒ Run(server, pk, m)@j ∧ j<i
- DY Objective oracle also checks DY security properties
  - Gather all the claims throughout traces executions at the PUT
  - Check all the DY security properties (where terms are concretized to bitstrings)

tlspuffin Implementation

tlspuffin: a full-fledge DY fuzzer

tlspuffin: a full-fledge DY fuzzer

- Open-source project written in Rust (16k LoC) (tlspuffin on Github)

tlspuffin: a full-fledge DY fuzzer

- Open-source project written in Rust (16k LoC) (tlspuffin on Github)

- Built on LibAFL, a modular library to build fuzzers (+ new/custom components⭐)

tlspuffin: a full-fledge DY fuzzer

- Open-source project written in Rust (16k LoC) (tlspuffin on Github)
- Built on LibAFL, a modular library to build fuzzers (+ new/custom components⭐)
- In-memory buffers, delightfully parallel, fast (700 execs/s/core)

tlspuffin: a full-fledge DY fuzzer

- Open-source project written in Rust (16k LoC) (tlspuffin on Github)
- Built on LibAFL, a modular library to build fuzzers (+ new/custom components⭐)
- In-memory buffers, delightfully parallel, fast (700 execs/s/core)
- For TLS: 189 function symbols, harnessed PUTs: OpenSSL, WolfSSL, LibreSSL

tlspuffin: a full-fledge DY fuzzer

- Open-source project written in Rust (16k LoC) (tlspuffin on Github)
- Built on LibAFL, a modular library to build fuzzers (+ new/custom components⭐)
- In-memory buffers, delightfully parallel, fast (700 execs/s/core)
- For TLS: 189 function symbols, harnessed PUTs: OpenSSL, WolfSSL, LibreSSL
- Beyond fuzzing: Connect to a PUT through TCP (easier to connect to new PUTs)
  + Traces are: executable, serializable, pretty-printable (as trees), concretizable (for PoC)

tlspuffin: a full-fledge DY fuzzer

- Open-source project written in Rust (16k LoC) (tlspuffin on Github)
- Built on LibAFL, a modular library to build fuzzers (+ new/custom components⭐)
- In-memory buffers, delightfully parallel, fast (700 execs/s/core)
- For TLS: 189 function symbols, harnessed PUTs: OpenSSL, WolfSSL, LibreSSL
- Beyond fuzzing: Connect to a PUT through TCP (easier to connect to new PUTs)
  + Traces are: executable, serializable, pretty-printable (as trees), concretizable (for PoC)
- Optimizations: fragmented output, queries for variables, transcript extraction

Puffin Modular Architecture

Puffin Modular Architecture

Puffin

- Terms (based on term signature)
- Traces + Domain-Specific Language
- Mutations
- Objective oracle (based on DY properties)
- Fuzzing-loop, CLI with all commands

Puffin Modular Architecture

Puffin

- Terms (based on term signature)
- Traces + Domain-Specific Language
- Mutations
- Objective oracle (based on DY properties)
- Fuzzing-loop, CLI with all commands

Once-for-all
6k LoC

tlspuffin

- Term signature
- Mapper: term → bitstrings
- DY security properties
- Seed corpus with DSL

Per protocol (here TLS)
Mapper most difficult
8k LoC

Puffin Modular Architecture

Puffin

- Terms (based on term signature)
- Traces + Domain-Specific Language
- Mutations
- Objective oracle (based on DY properties)
- Fuzzing-loop, CLI with all commands

Once-for-all
6k LoC

tlspuffin

- Term signature
- Mapper: term → bitstrings
- DY security properties
- Seed corpus with DSL

Per protocol (here TLS)
Mapper most difficult
8k LoC

OpenSSL Harness

- Executor interface
- Claim extractions

Per PUT, quite lightweight
+ TCP mode
500 LoC

Puffin Modular Architecture

Puffin

- Terms (based on term signature)
- Traces + Domain-Specific Language
- Mutations
- Objective oracle (based on DY properties)
- Fuzzing-loop, CLI with all commands

Once-for-all
6k LoC

tlspuffin

- Term signature
- Mapper: term → bitstrings
- DY security properties
- Seed corpus with DSL

Per protocol (here TLS)
Mapper most difficult
8k LoC

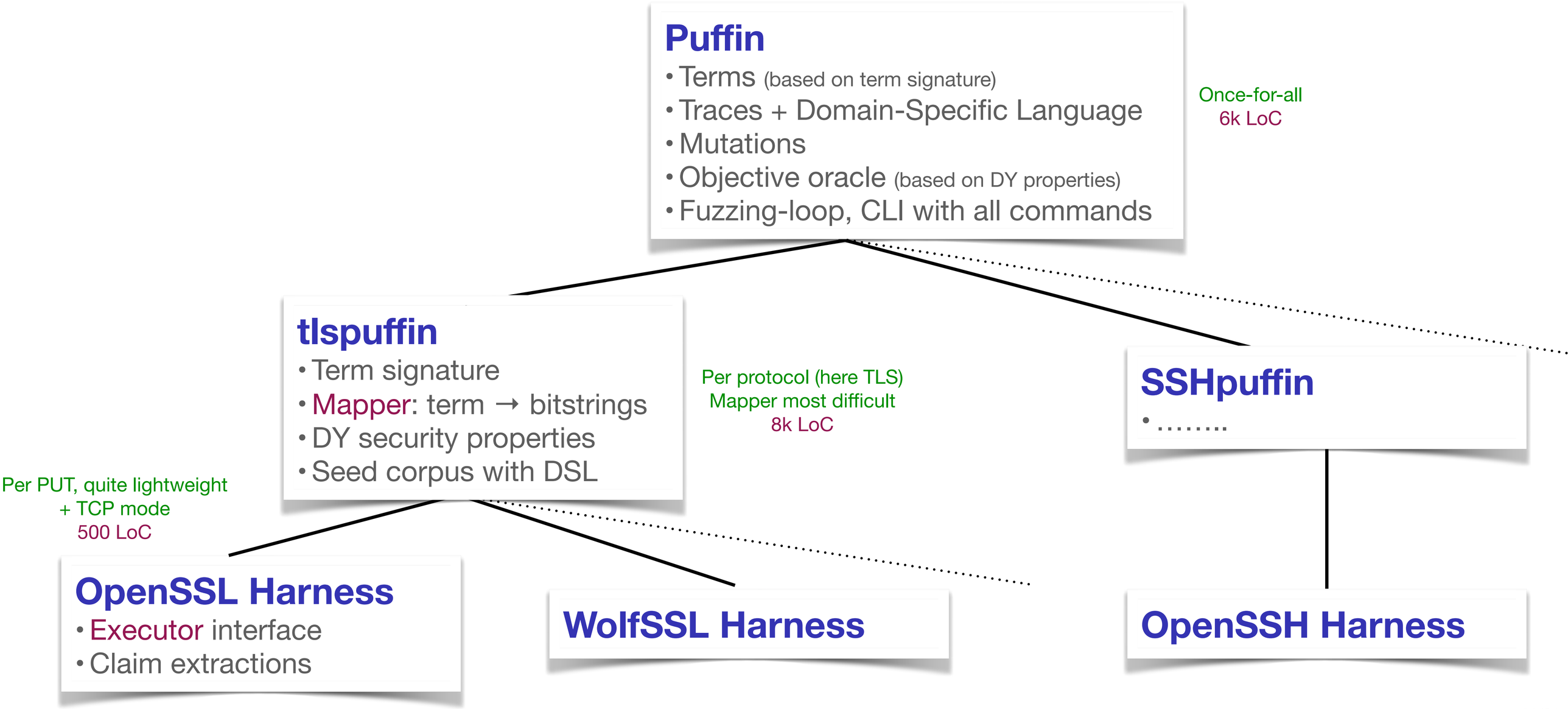
Per PUT, quite lightweight
+ TCP mode
500 LoC

OpenSSL Harness

- Executor interface
- Claim extractions

WolfSSL Harness

Puffin Modular Architecture



tlspuffin Results

tlspuffin findings 

tlspuffin findings 

- We selected a small benchmark suite: recent logical attacks found on OpenSSL (most used) and WolfSSL (IoT)

tlspuffin findings 

- We selected a small benchmark suite: recent logical attacks found on OpenSSL (most used) and WolfSSL (IoT)
- Found by tlspuffin in hours or seconds (SKIP), systematic reproducibility!

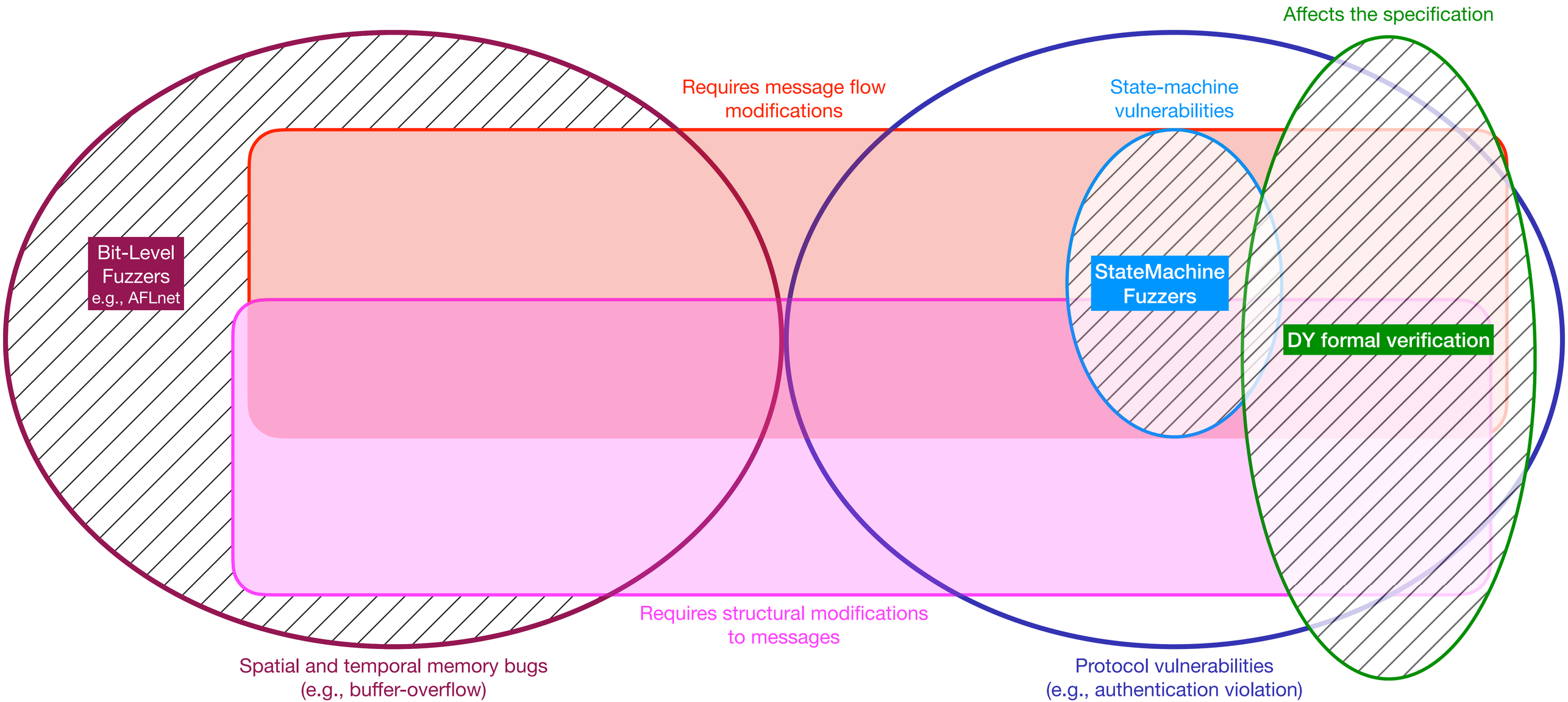
CVE ID	AKA	CVSS	Type	New	Version	TLS
2021-3449	**SDOS1**	5.9	Server DoS, M	✗	1.1.1j	1.2
2022-25638	**SIG**	6.5	Auth. Bypass, P	✗	5.1.0	1.3
2022-25640	**SKIP**	7.5	Auth. Bypass, P	✗	5.1.0	1.3

tlspuffin findings 

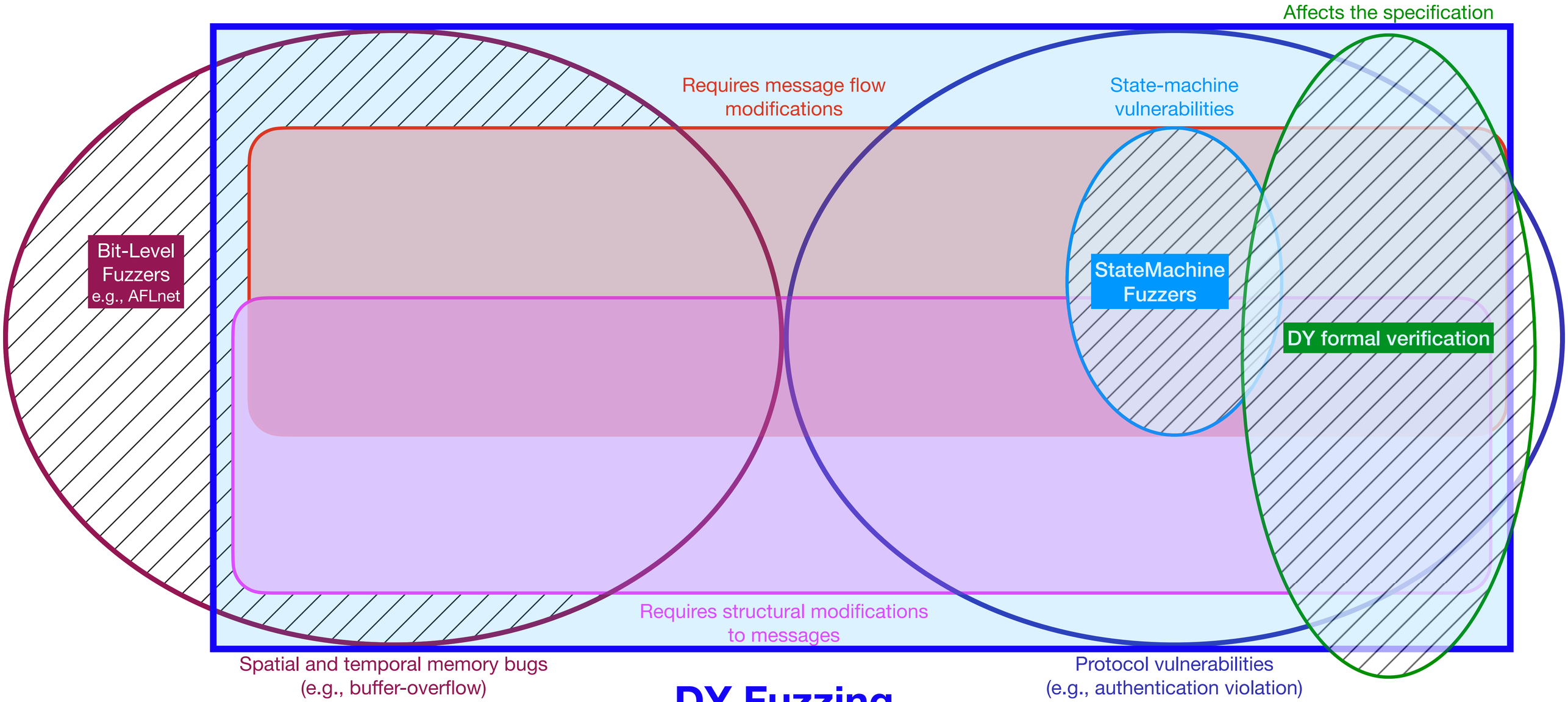
- We selected a small benchmark suite: recent logical attacks found on OpenSSL (most used) and WolfSSL (IoT)
- Found by tlspuffin in hours or seconds (SKIP), systematic reproducibility!
- We ran fuzzing campaigns on the harnessed PUTs and found 4 new CVEs
  ☞Not found by other fuzzers

CVE ID	AKA	CVSS		Type	New	Version	TLS
2021-3449	**SDOS1**	5.9		Server DoS, M	✗	1.1.1j	1.2
2022-25638	**SIG**	6.5		Auth. Bypass, P	✗	5.1.0	1.3
2022-25640	**SKIP**	7.5		Auth. Bypass, P	✗	5.1.0	1.3
2022-38152	**SDOS2**	7.5	!	Server DoS, M	✓	5.4.0	1.3
2022-38153	**CDOS**	5.9		Client DoS, M	✓	5.3.0	1.2
2022-39173	**BUF**	7.5	!	Server DoS, M	✓	5.5.0	1.3
2022-42905	**HEAP**	9.1	!!!	Info. Leak, M	✓	5.5.0	1.3

Retrospective of TLS Failures https:// 2014-2022

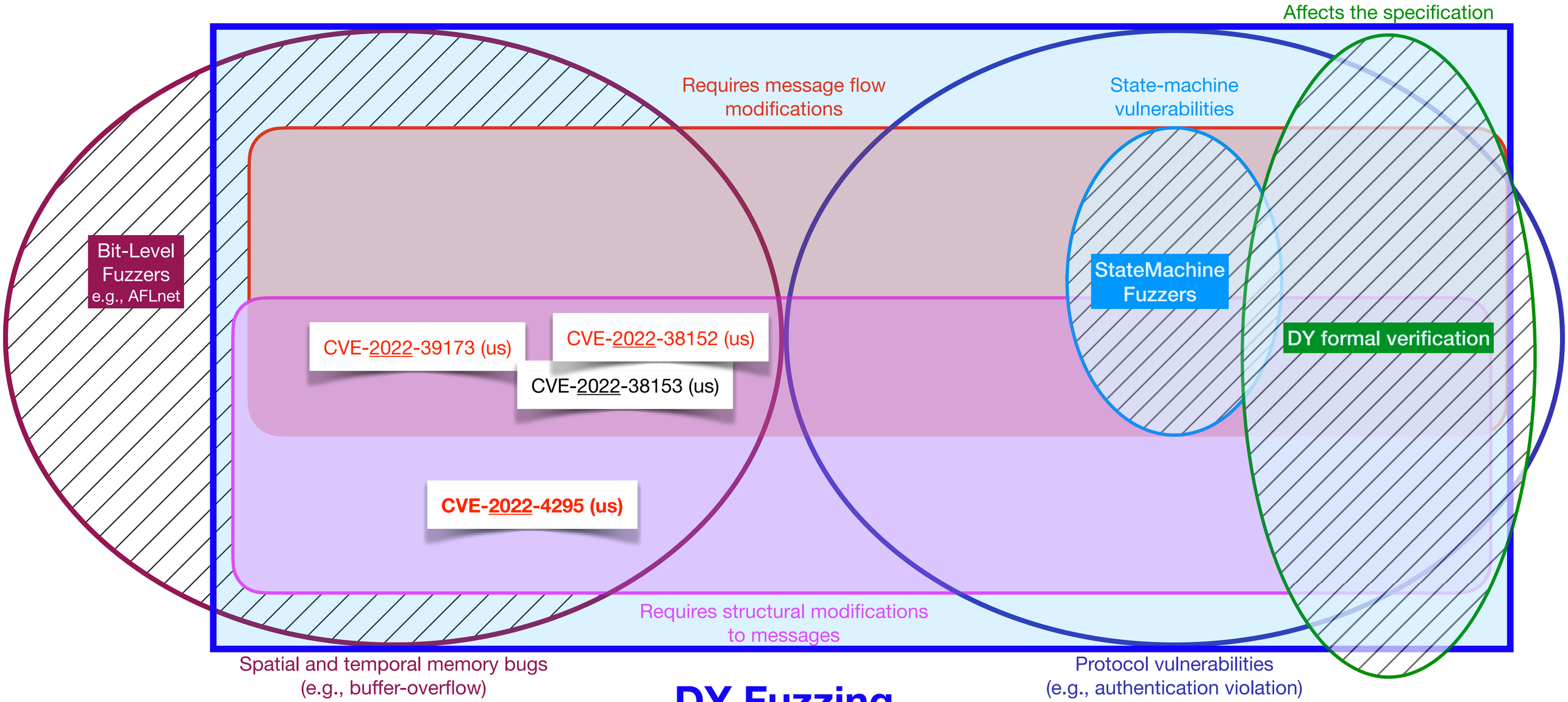


Retrospective of TLS Failures https:// 2014-2022



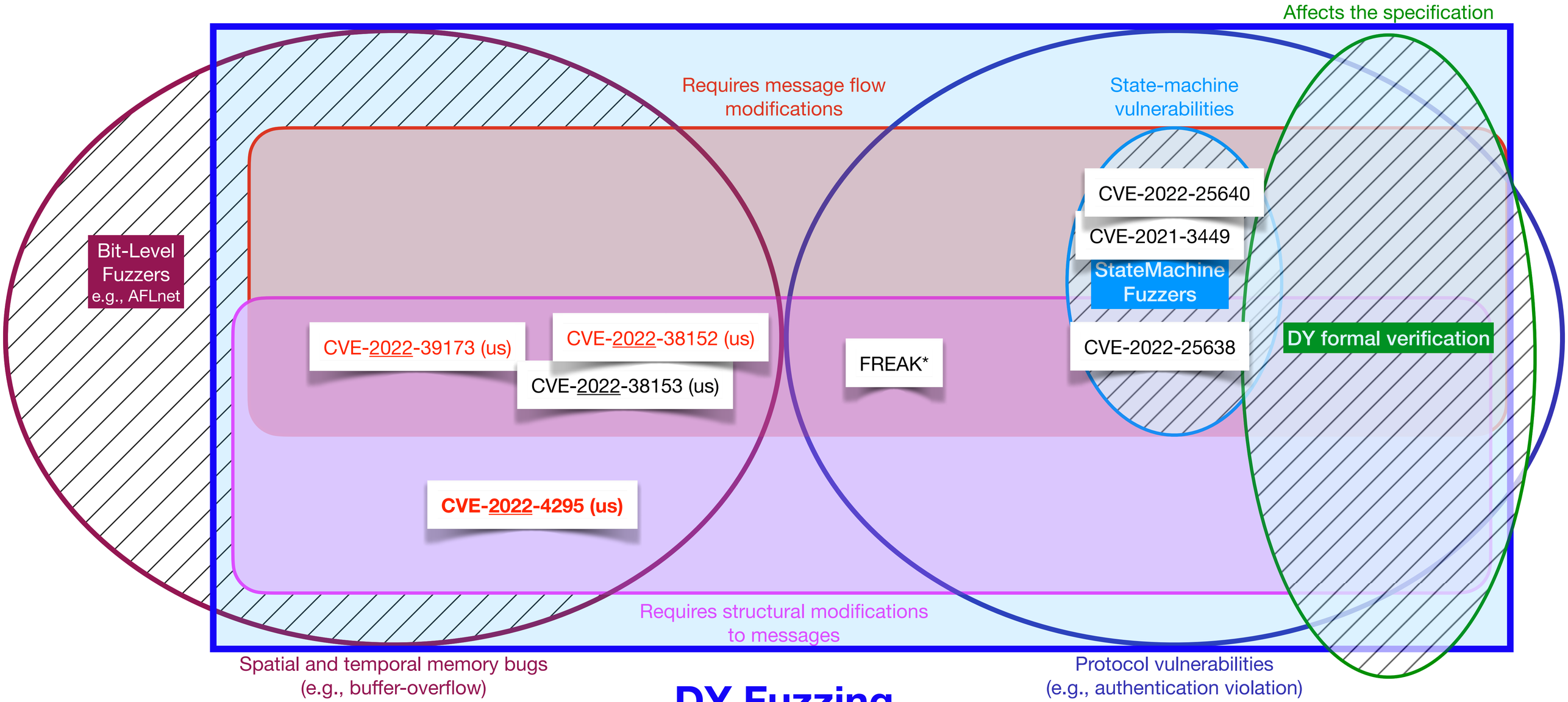
DY Fuzzing

Retrospective of TLS Failures https:// 2014-2022



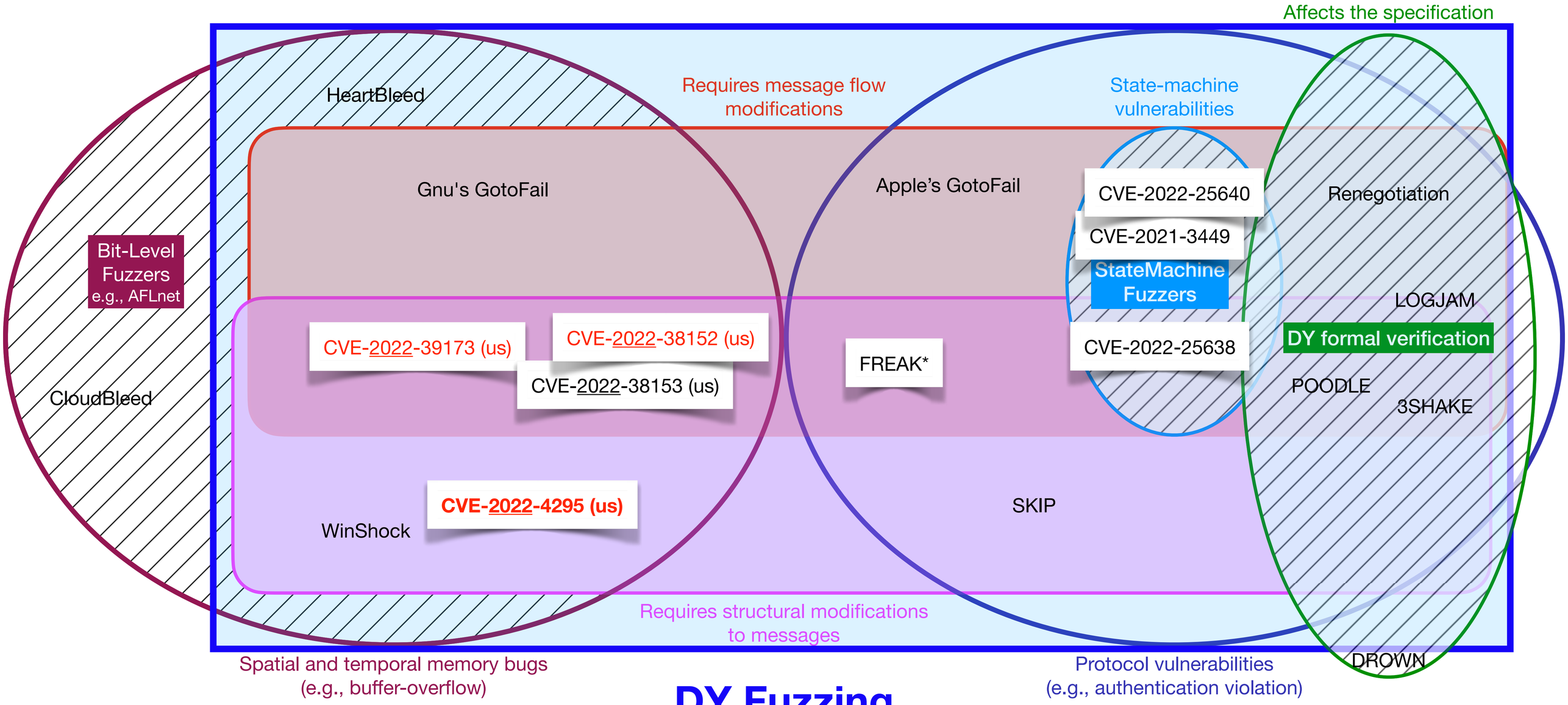
DY Fuzzing

Retrospective of TLS Failures https:// 2014-2022



DY Fuzzing

Retrospective of TLS Failures https:// 2014-2022



DY Fuzzing

Bug triaging

1. Objective traces are stored on disk during fuzzing
2. Execute against clean slate WolfSSL through TCP
3. Plot the trace, inspect the attacker terms, could modify and re-execute

   ☞ Understand the attack requirements

4. gdb/ldb tlspuffin+WolfSSL execute trace (action-by-action, step-by-step)

   ☞ Understand the attack root causes

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)
2. Forges a malicious ClientHello([c;..;c]) message such that

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)
2. Forges a malicious ClientHello([c;..;c]) message such that
   (a) it **resumes previous session with PSK** (**needs to apply decrypt, hash, signature)** and
   (b) It has as list of supported cipher suites with **duplicates of** c (say n times)

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)
2. Forges a malicious ClientHello([c;..;c]) message such that
   (a) it **resumes previous session with PSK** (**needs to apply decrypt, hash, signature)** and
   (b) It has as list of supported cipher suites with **duplicates of** c (say n times)

☞ Server calls refineSuites to update suitesS (ciphers offered both by client and server) bc. of **resumption**

☞ **Flaw 1**: **actually** computes « multiset-intersection » so suitesS will contain **duplicates of** c (say k times)

☞ No big deal because suitesS initially had no duplicate so: k = n ≤ |suitesC| ≤ MAX_SZ = 150

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)
2. Forges a malicious ClientHello([c;..;c]) message such that
   (a) it **resumes previous session with PSK (needs to apply decrypt, hash, signature)** and
   (b) It has as list of supported cipher suites with **duplicates of** c (say n times)

☞ Server calls refineSuites to update suitesS (ciphers offered both by client and server) bc. of **resumption**

☞ **Flaw 1**: **actually** computes « multiset-intersection » so suitesS will contain **duplicates of** c (say k times)

☞ No big deal because suitesS initially had no duplicate so: k = n ≤ |suitesC| ≤ MAX_SZ = 150

How WolfSSL implements ∩ with refineSuites(suitesC)@tls13.c:4355

```
// suitesS initially with offered suites, MAX_SZ allocated
byte suites[MAX_SZ]; int suiteSz = 0; // supposed to compute suitesS ∩ suitesC

for (i = 0; i < suitesS.size; i += 1) {
      for (j = 0; j < suitesC.size; j += 1) {    // suitesC.size <= MAX_SZ
            if (suitesS->suites[i] == suitesC->suites[j]) {
                suites[suiteSz++] = suitesC->suites[j]; } } }
XMEMCPY(suitesS, &suites, sizeof(suites));
```

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)
2. Forges a malicious ClientHello([c;..;c]) message such that
   (a) it **resumes previous session with PSK** (**needs to apply decrypt, hash, signature)** and
   (b) It has as list of supported cipher suites with **duplicates of** c (say n times)

☞ Server calls refineSuites to update suitesS (ciphers offered both by client and server) bc. of **resumption**

☞ **Flaw 1**: **actually** computes « multiset-intersection » so suitesS will contain **duplicates of** c (say k times)

☞ No big deal because suitesS initially had no duplicate so: k = n ≤ |suitesC| ≤ MAX_SZ = 150

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)
2. Forges a malicious ClientHello([c;..;c]) message such that
   (a) it **resumes previous session with PSK (needs to apply decrypt, hash, signature)** and
   (b) It has as list of supported cipher suites with **duplicates of** c (say n times)

☞ Server calls refineSuites to update suitesS (ciphers offered both by client and server) bc. of **resumption**

☞ **Flaw 1**: **actually** computes « multiset-intersection » so suitesS will contain **duplicates of** c (say k times)

☞ No big deal because suitesS initially had no duplicate so: $k = n \leq |suitesC| \leq MAX\_SZ = 150$

   (c) Is **ill-formed** and **will be rejected but *late*** (after call to refineSuites), mess with supportGroupExtension

☞ Server rejects it and sends a HelloRetryRequest but

☞ **Flaw 2**: **side-effects** of refineSuites **are not reverted**

☞ **From now on,** refineSuites invariant is broken: suitesS contains n duplicates of c

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)
2. Forges a malicious ClientHello([c;..;c]) message such that
   (a) it **resumes previous session with PSK** (**needs to apply decrypt, hash, signature)** and
   (b) It has as list of supported cipher suites with **duplicates of** c (say n times)

☞ Server calls refineSuites to update suitesS (ciphers offered both by client and server) bc. of **resumption**

☞ **Flaw 1**: **actually** computes « multiset-intersection » so suitesS will contain **duplicates of** c (say k times)

☞ No big deal because suitesS initially had no duplicate so: $k = n \leq |suitesC| \leq MAX\_SZ = 150$

   (c) Is **ill-formed** and **will be rejected but *late*** (after call to refineSuites), mess with supportGroupExtension

☞ Server rejects it and sends a HelloRetryRequest but

☞ **Flaw 2**: **side-effects** of refineSuites **are not reverted**

☞ **From now on,** refineSuites invariant is broken: suitesS contains n duplicates of c

3. Send ClientHello([c;..;c]) again, refineSuites is called again, the resulting buffer suites that contains $k^2 = n^2$ ciphers c is copied into suitesS

☞ For n = 13, we already overwrite the suitesS buffer allocated on MAX_ciphers_list_length = 150

Root causes of CVE-2022-39173 (WolfSSL, CVSS high)

1. Attacker acting as client performs a full TLS handshake, establishing a Pre-Shared-Key (PSK)
2. Forges a malicious ClientHello([c;..;c]) message such that
   (a) it **resumes previous session with PSK (needs to apply decrypt, hash, signature)** and
   (b) It has as list of supported cipher suites with **duplicates of** c (say n times)

☞ Server calls refineSuites to update suitesS (ciphers offered both by client and server) bc. of **resumption**

☞ **Flaw 1**: **actually** computes « multiset-intersection » so suitesS will contain **duplicates of** c (say k times)

☞ No big deal because suitesS initially had no duplicate so: k = n ≤ |suitesC| ≤ MAX_SZ = 150

   (c) Is **ill-formed** and **will be rejected but *late*** (after call to refineSuites), mess with supportGroupExtension

An overflow on the stack of max 44700 bytes (controlled by n).

- ☞ **Therefore, large portions of the stack can get overwritten, including return addresses (confirmed)**
- ☞ **Potential RCE (unconfirmed)**
- ☞ **Potential for negotiating ciphers that server should reject (downgrade)**

☞ For n = 13, we already overwrite the suitesS buffer allocated on MAX_ciphers_list_length = 150

DY Fuzzing Future Work

Future Work - Evaluation

Future Work - Evaluation

- tlspuffin always found the new CVEs
- state-of-the art competitive fuzzers never found any of them

We can explain this with qualitative evidences but quantitative evidences are hard to obtain

Future Work - Evaluation

- tlspuffin always found the new CVEs
- state-of-the art competitive fuzzers never found any of them

We can explain this with qualitative evidences but quantitative evidences are hard to obtain

- **Code-coverage** is a poor metric

A statement reached from an attack state is similarly counted as if reached from the happy flow

---

E.g., client accepting a legitimate server's certificate $=_{coverage}$ *accepting illegitimate cert.*

Future Work - Evaluation

- tlspuffin always found the new CVEs
- state-of-the art competitive fuzzers never found any of them

We can explain this with qualitative evidences but quantitative evidences are hard to obtain

- **Code-coverage** is a poor metric

A statement reached from an attack state is similarly counted as if reached from the happy flow

---

E.g., client accepting a legitimate server's certificate $=_{coverage}$ *accepting illegitimate cert.*

- **Yet, some insights by manual analysis of the diff-coverage** (tlspuffin vs. AFLnet)
  - tlspuffin explores more extensions requiring structured messages approach (crypto)
    (e.g., mutations UNDER encryption/signature)
  - Other fuzzers beat tlspuffin code-coverage for discovering some functionalities in ClientHello (e.g., discover a lot more ciphers yet without being able to then use them)

Future Work (cont.)

Future Work (cont.)

DY coverage: code-coverage is currently a bottleneck (prone to exhaustion)

→ Need for a domain-specific DY-based notion of coverage
Hitting the same code with different adversarial behaviors should not be considered the same

→ Combine with a proxy for how close a trace is to an attack trace
Could be useful to incentivize better term generation and some attack scenarios

→ Combine and find a balance with code-based coverage (specified vs. implemented functionalities)

Future Work (cont.)

DY coverage: code-coverage is currently a bottleneck (prone to exhaustion)

→ Need for a domain-specific DY-based notion of coverage
Hitting the same code with different adversarial behaviors should not be considered the same

→ Combine with a proxy for how close a trace is to an attack trace
Could be useful to incentivize better term generation and some attack scenarios

→ Combine and find a balance with code-based coverage (specified vs. implemented functionalities)

Improved objective oracle

- Differential fuzzing: save t as objective when WolfSSL(t) ≠ OpenSSL(t)

Future Work (cont.)

DY coverage: code-coverage is currently a bottleneck (prone to exhaustion)

→ Need for a domain-specific DY-based notion of coverage
Hitting the same code with different adversarial behaviors should not be considered the same

→ Combine with a proxy for how close a trace is to an attack trace
Could be useful to incentivize better term generation and some attack scenarios

→ Combine and find a balance with code-based coverage (specified vs. implemented functionalities)

Improved objective oracle

- Differential fuzzing: save t as objective when WolfSSL(t) ≠ OpenSSL(t)
- Or extend the oracle: more compromise scenarios, secrecy (abstraction, deduction?), privacy (approx.?), functional correctness / a model

Future Work (cont.)

Future Work (cont.)

- Combine DY fuzzing with bit-level fuzzing (WIP): reach « deep states » with DY attacker and then smash with some bit-level mutations

Future Work (cont.)

- Combine DY fuzzing with bit-level fuzzing (WIP): reach « deep states » with DY attacker and then smash with some bit-level mutations

- DY-based concolic testing: use DY verifiers to synthesize test cases that pass “complex” conditions

Future Work (cont.)

- Combine DY fuzzing with bit-level fuzzing (WIP): reach « deep states » with DY attacker and then smash with some bit-level mutations

- DY-based concolic testing: use DY verifiers to synthesize test cases that pass “complex” conditions

- Apply DY fuzzing to more protocols and PUTs (e.g., SChannel, WPA, TelCo)

Future Work (cont.)

- Combine DY fuzzing with bit-level fuzzing (WIP): reach « deep states » with DY attacker and then smash with some bit-level mutations

- DY-based concolic testing: use DY verifiers to synthesize test cases that pass “complex” conditions

- Apply DY fuzzing to more protocols and PUTs (e.g., SChannel, WPA, TelCo)

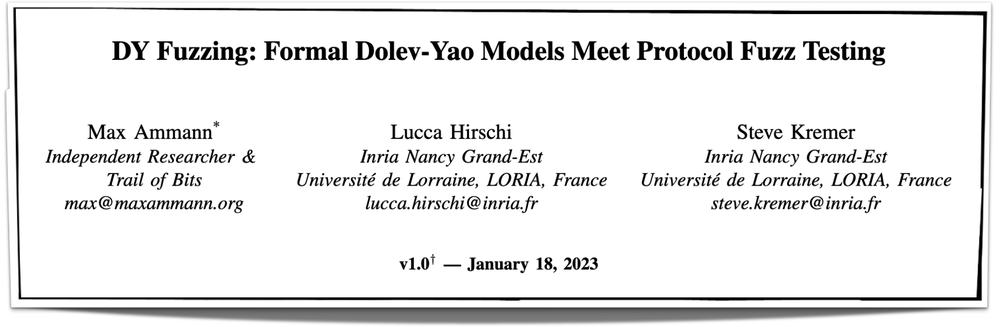
Long-Term
- (Partially) Automate Mapper and Harness → PUT-agnostic DY fuzzer
- Model extraction
- Connect further with DY verifiers (ProVerif, Tamarin, Sapic+)

Summary of Contributions

Summary of Contributions

1. A new approach to fuzzing cryptographic protocols connecting the DY formal approach with fuzzing
   → captures for the first time the class of logical attacks / DY attacker
2. DY Fuzzing design specification
3. tlspuffin: full-fledged, modular, efficient DY fuzzer implementation for TLS
4. Evaluate tlspuffin on TLS libraries:
   - (re)found seven vulnerabilities
   - including four new ones (one critical, two high, and one medium)

Preprint IACR 2023/057



DY Fuzzing: Formal Dolev-Yao Models Meet Protocol Fuzz Testing

Max Ammann*
Independent Researcher & Trail of Bits
max@maxammann.org

Lucca Hirschi
Inria Nancy Grand-Est
Université de Lorraine, LORIA, France
lucca.hirschi@inria.fr

Steve Kremer
Inria Nancy Grand-Est
Université de Lorraine, LORIA, France
steve.kremer@inria.fr

v1.0† — January 18, 2023

Project ANR JCJC
→ Looking for student/postdocs/engineers

AAPG2022	PROTOFUZZ		JCJC
Coordinated by	Lucca Hirschi	36 months	280 805€
Axe E.1 : Fondements du numérique : informatique, automatique, traitement du signal			

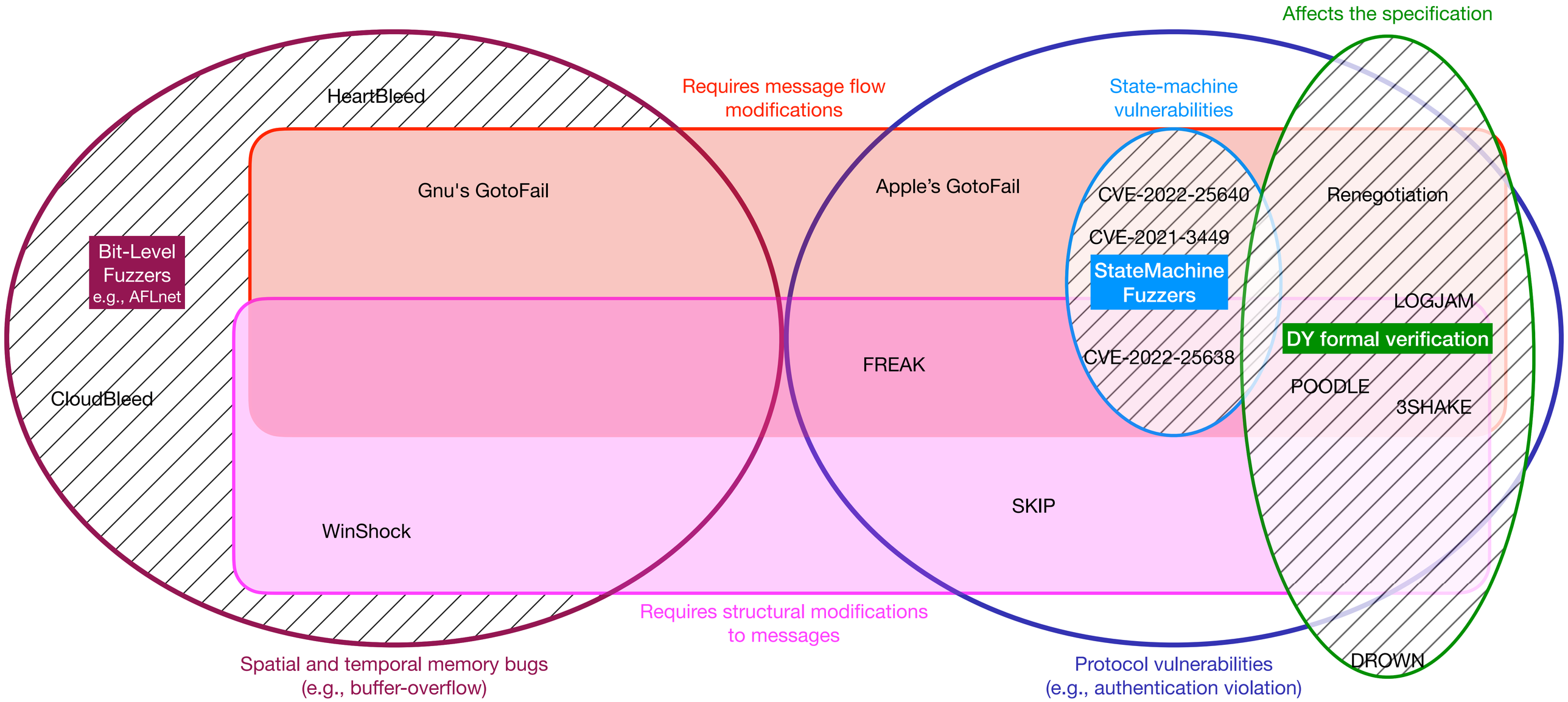
PROTOFUZZ: Cryptographic Protocol Logic Fuzz Testing

Formal Verification Meets Fuzz Testing

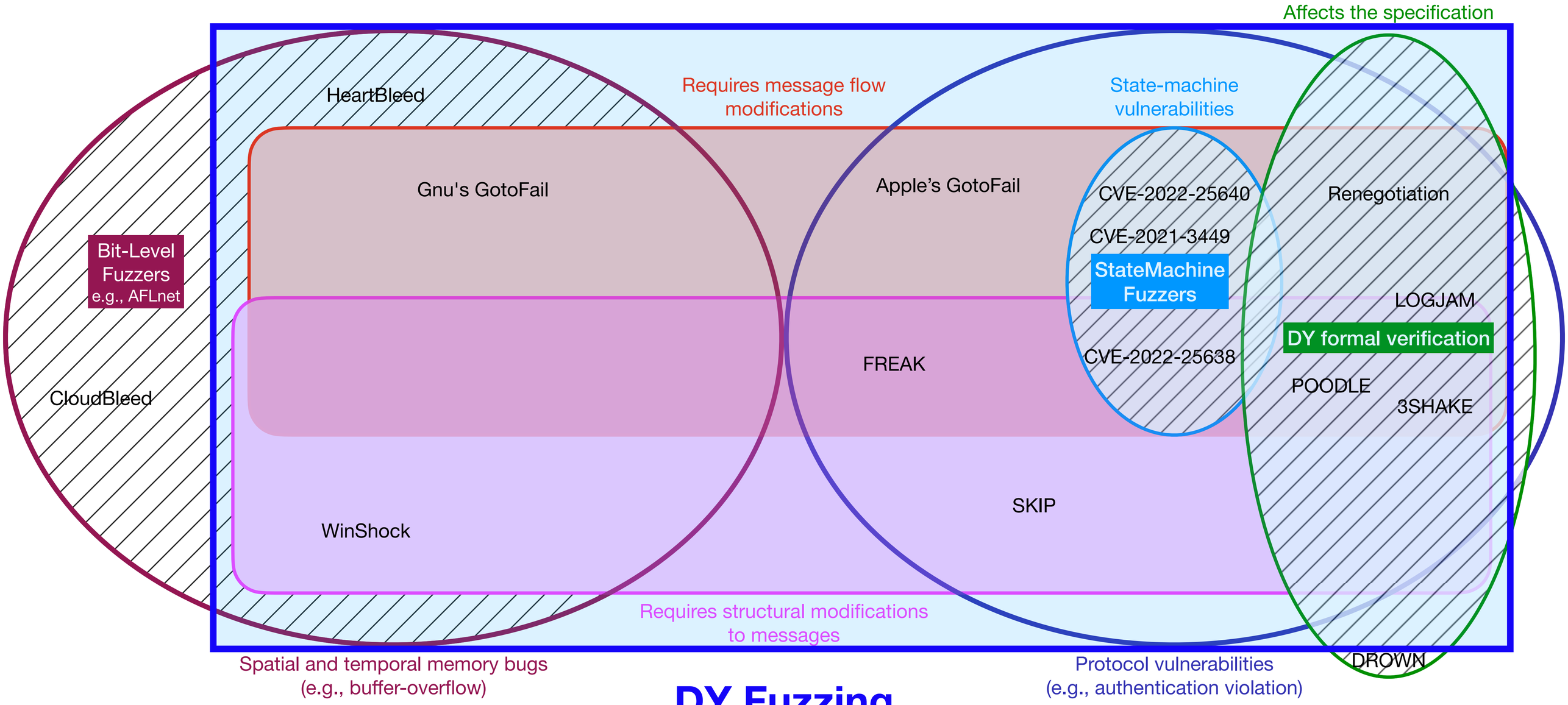
Consortium: PESTO (Inria Nancy)

Backup Slides

Retrospective of TLS Failures https:// 2019-2022

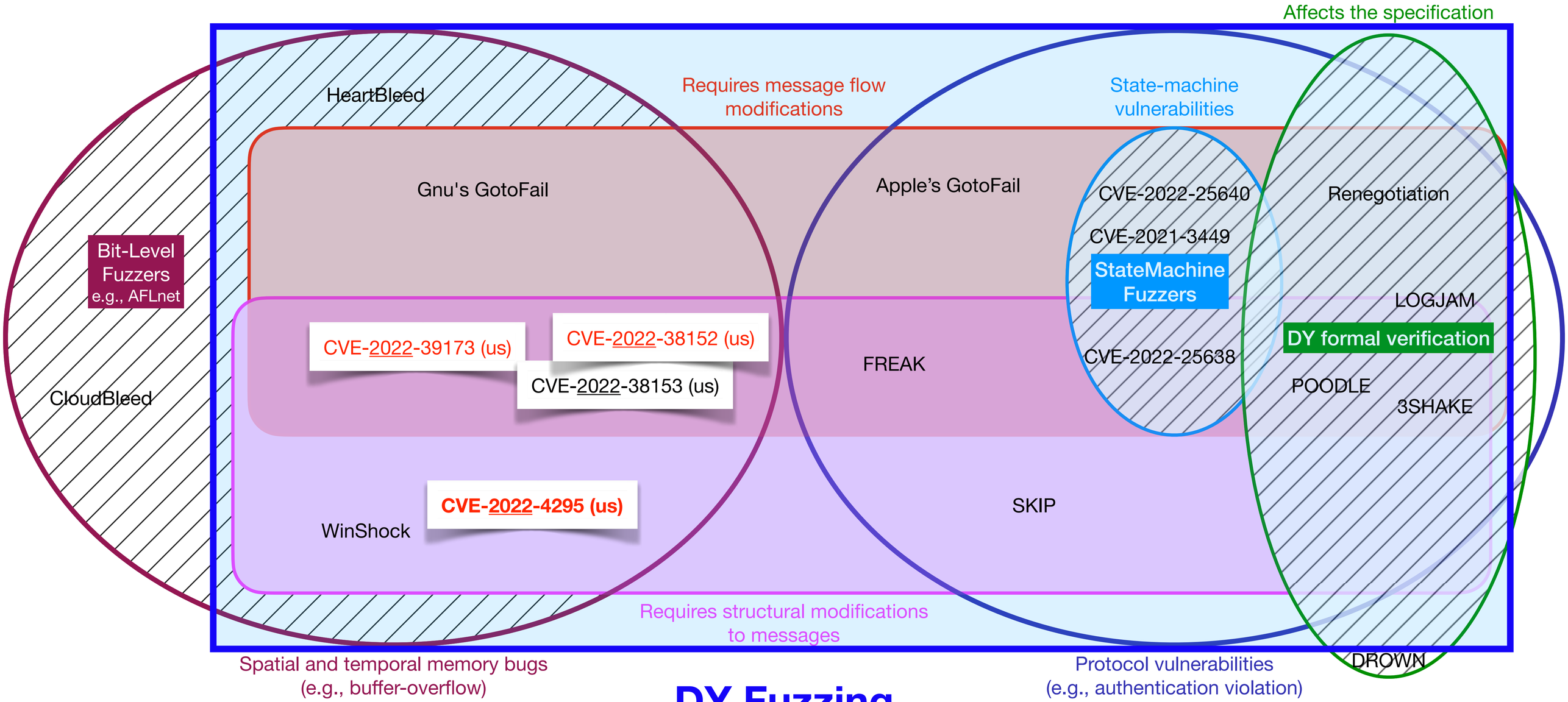


Retrospective of TLS Failures https:// 2019-2022



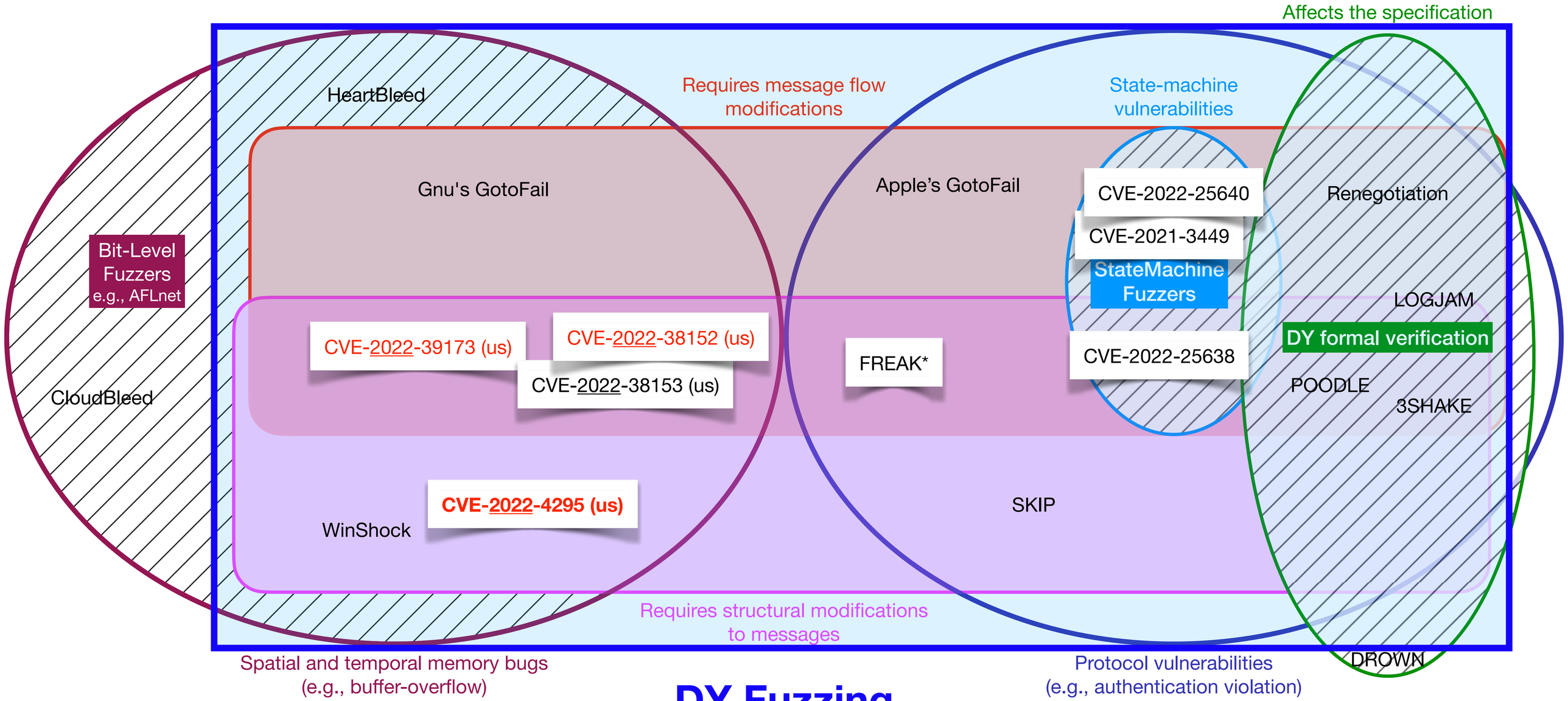
DY Fuzzing

Retrospective of TLS Failures https:// 2019-2022



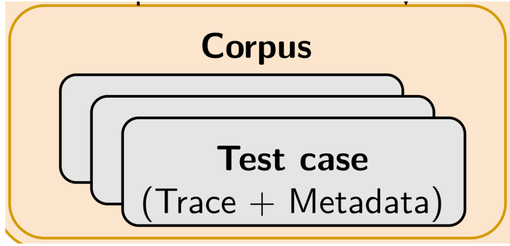
DY Fuzzing

Retrospective of TLS Failures https:// 2019-2022

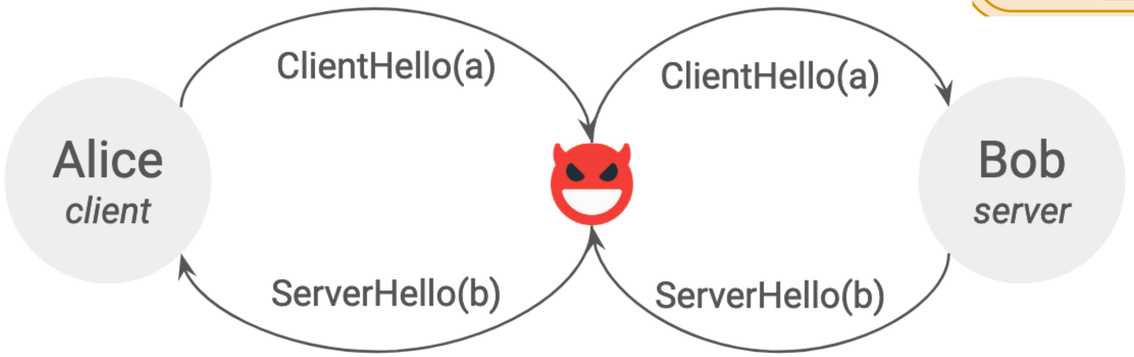


DY Fuzzing

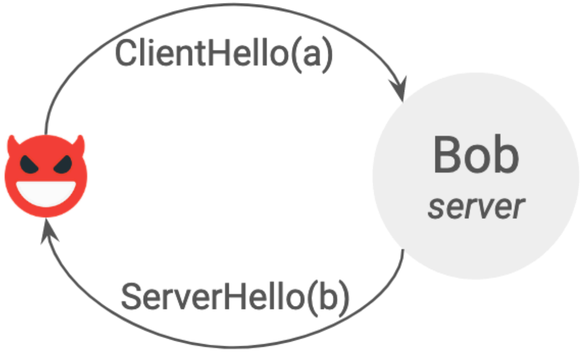
Seed Corpus



MITM Happy Flow



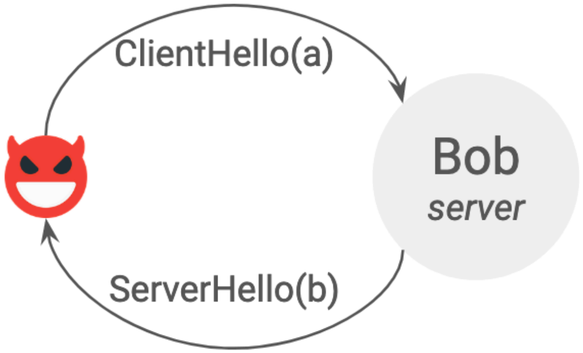
Client Attacker
(Happy flow)



Session Resumption
(Happy flow)

Similar to Client attacker, but performs a second handshake

2x



Tlspuffin Terms Domain-Specific Language

```
let rsa_certificate = term! {
  fn_certificate13(...)
};

let certificate = term! {
  fn_encrypt_handshake(
    (@certificate_rsa),
    (fn_sh_transcript(((server, 0)))),
    (fn_server_share(((server, 0)))),
    fn_seq_0,
    ...
  )
};
```

Tlspuffin Traces Domain-Specific Language

```
let recipe: Term = term! {
  fn_client_hello(
    fn_protocol_version12,
    fn_random,
    fn_session_id,
    fn_cipher_suites,
    fn_compressions,
    fn_client_extensions
  )
};
```

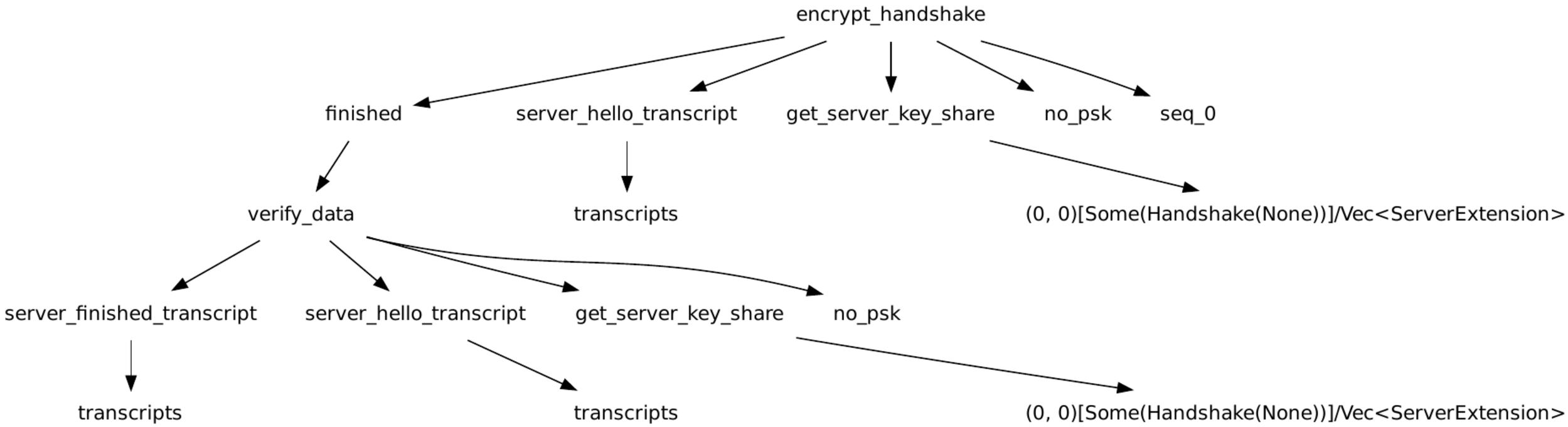
Domain Specific Language for declaring terms

```
let client_name = AgentName::new();
let server_name = client::next();
```

```
let steps: Vec<Steps> = vec![
  OutputAction::new_step(client_name),
  InputAction::new_step(
    server_name, recipe
  )
];
```

```
let trace: Trace = Trace {
  prior_traces: vec![],
  descriptors: vec![
    AgentDescriptor::
      new_client(client_name, V1_3),
    AgentDescriptor::
      new_server(server_name, V1_3)
    ],
  steps
};
```

Plotting Terms and Traces



tlspuffin: a full-fledge DY fuzzer

tlspuffin: a full-fledge DY fuzzer

- Open-source project written in Rust (16k LoC) (tlspuffin on Github)
- Built on LibAFL, a modular library to build fuzzers (+ new/custom components⭐)
- In-memory buffers, delightfully parallel, fast (700 execs/s/core)
- For TLS: 189 function symbols, harnessed PUTs: OpenSSL, WolfSSL, OpenSSL
- Beyond fuzzing: Connect to a PUT through TCP (easier to connect to new PUTs)
  + Traces are: executable, serializable, pretty-printable (as trees), concretizable (for PoC)
- Optimizations:
  - fragment outputs by extracting sub-messages → smaller terms
  - queries for accessing output variable access → more robust through mutations
  - automatic transcript extraction → much smaller terms, think < m,MAC(h(transcript),k) >