

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

[Max Ammann](#), Trail of Bits, USA
[Tom Gouvill](#), Inria, France
[Lucca Hirschi](#), Inria, France
[Steve Kremer](#), Inria, France
[Michael Mera](#), Inria, France



Workshop on the Analysis of Network Protocols
September 17, 2024

Secure Cryptographic Protocols

Cryptographic Protocols

Informal definition

concurrent program relying on **cryptography** to **secure** communications

Example: TLS using ECDH, SHA2, ... to establish secure channels

Security goals: confidentiality, integrity and authentication

Protocols: Signal, EMV (credit cards), e-voting, 5G authentication, RFID, etc.

Cryptographic Protocols

Informal definition

concurrent program relying on **cryptography** to **secure** communications

Example: TLS using ECDH, SHA2, ... to establish secure channels

Security goals: confidentiality, integrity and authentication

Protocols: Signal, EMV (credit cards), e-voting, 5G authentication, RFID, 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?

Memory Safety Vulnerabilities

⚡ HeartBleed

(e.g. buffer-overflow)

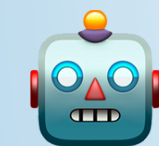
**Retrospective
of some TLS**  <https://>
Failures ⚡

Memory Safety Vulnerabilities

⚡ HeartBleed



Fuzzing



01
0010
1101

10
0010
1110

(e.g. buffer-overflow)

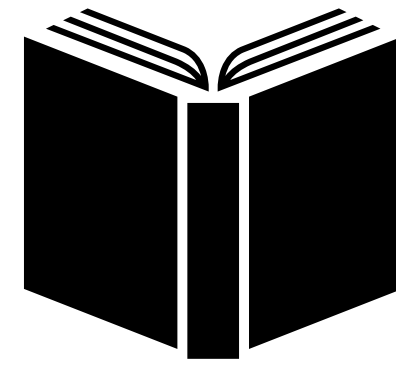
**Retrospective
of some TLS**  <https://>
Failures ⚡

State-of-the-Art Fuzzing

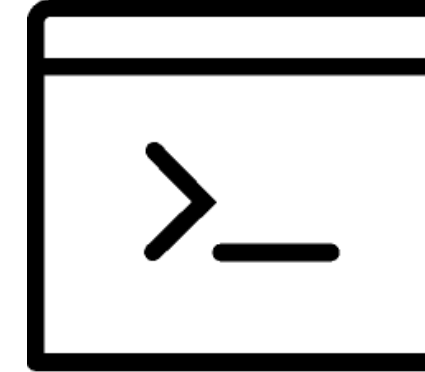
Limitations

Bit-level Fuzzing (AFL-like)

Corpus of test-cases

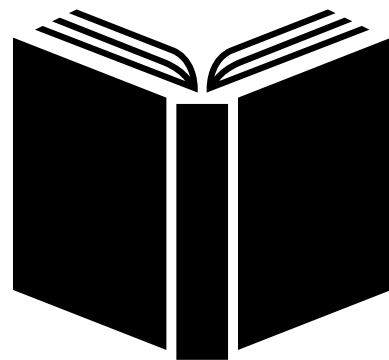


Program Under Test

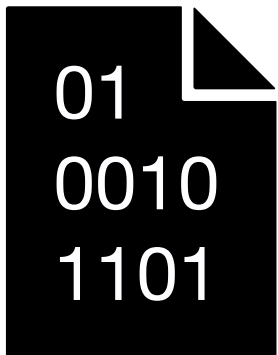


Bit-level Fuzzing (AFL-like)

Corpus of test-cases

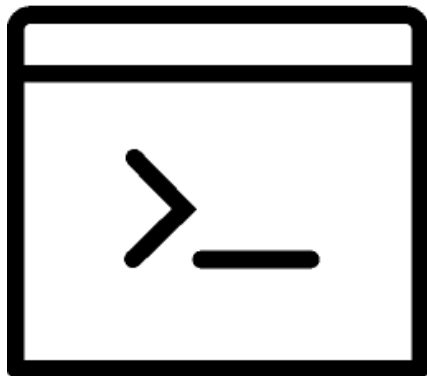


Pick a test-case

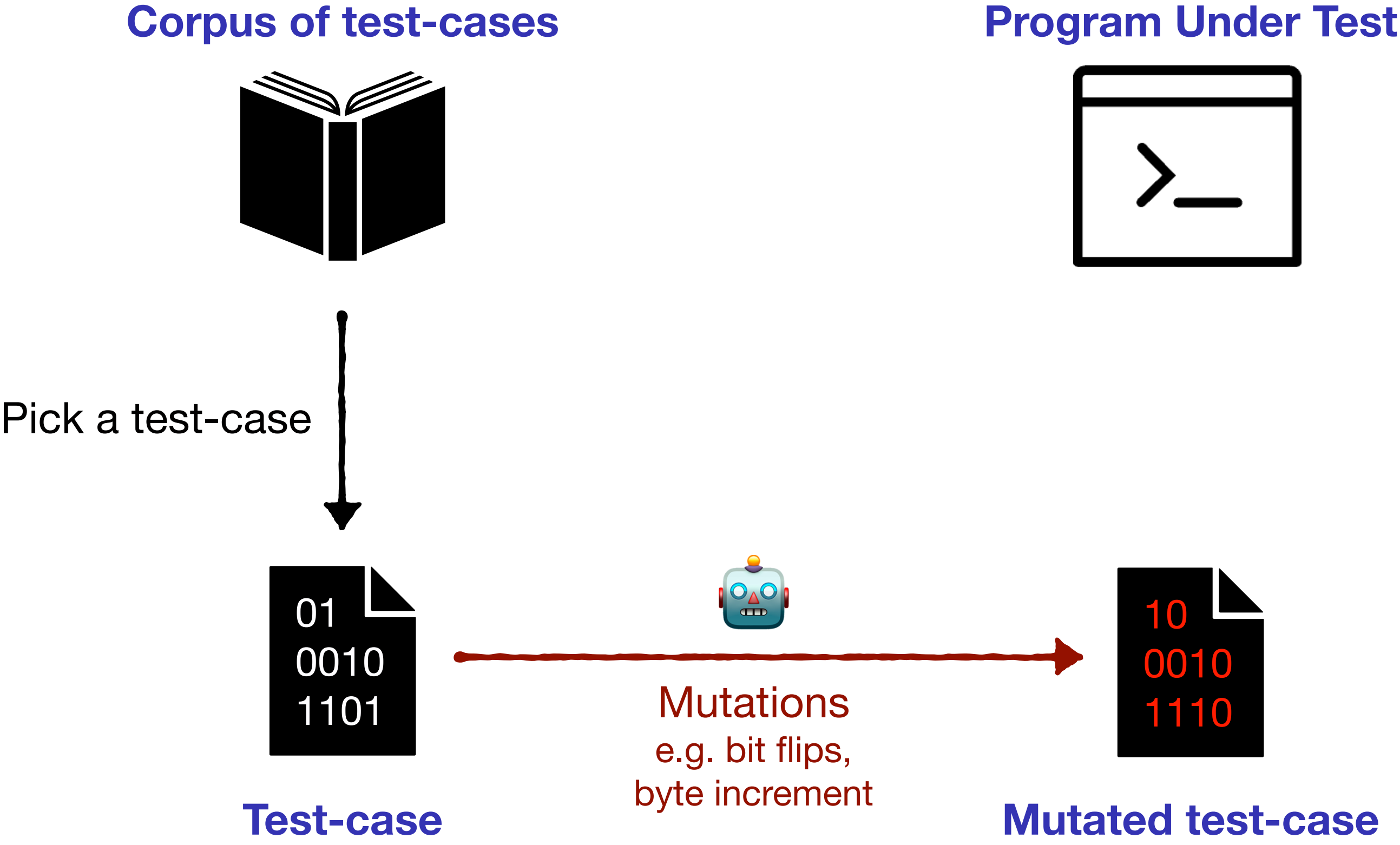


Test-case

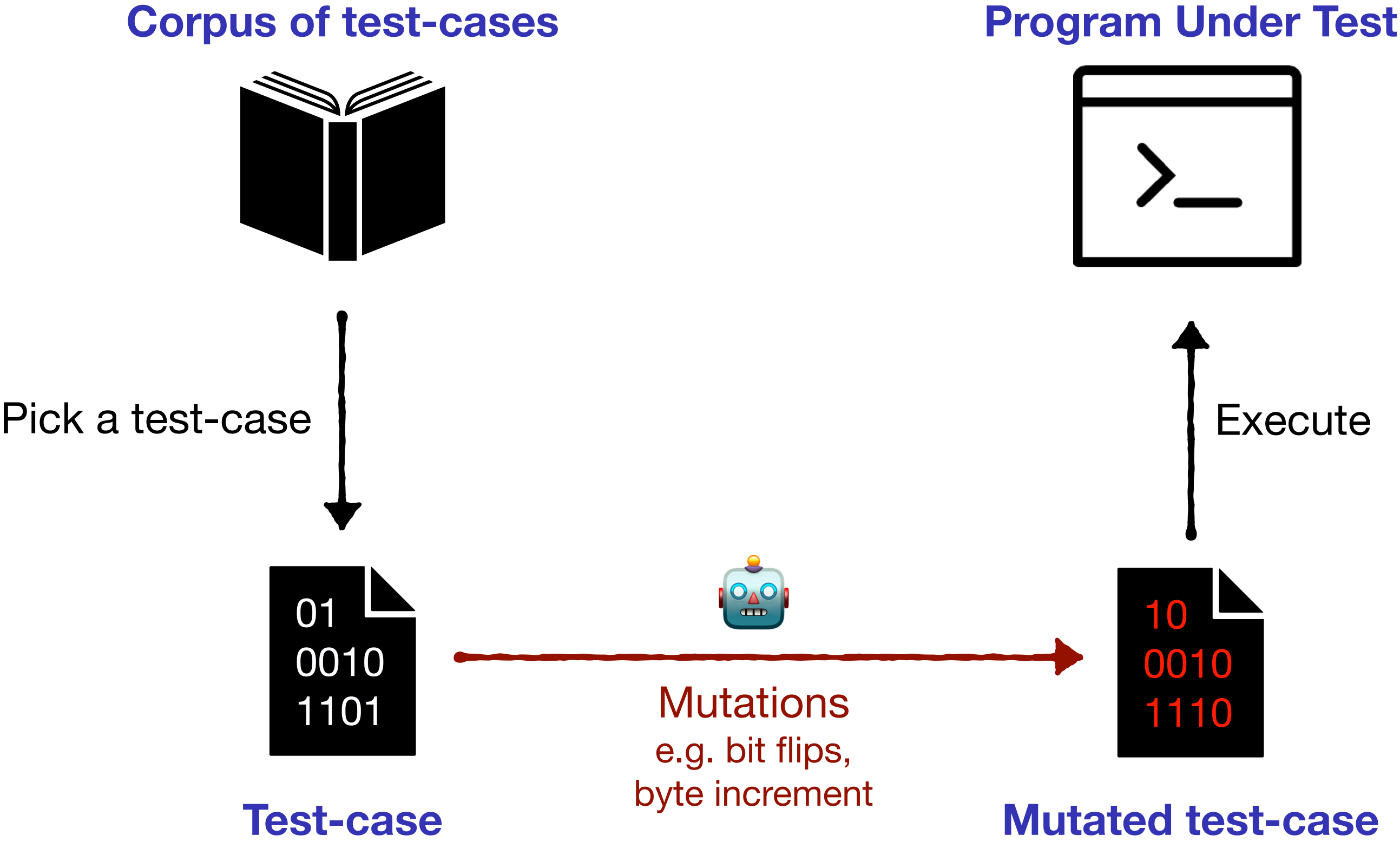
Program Under Test



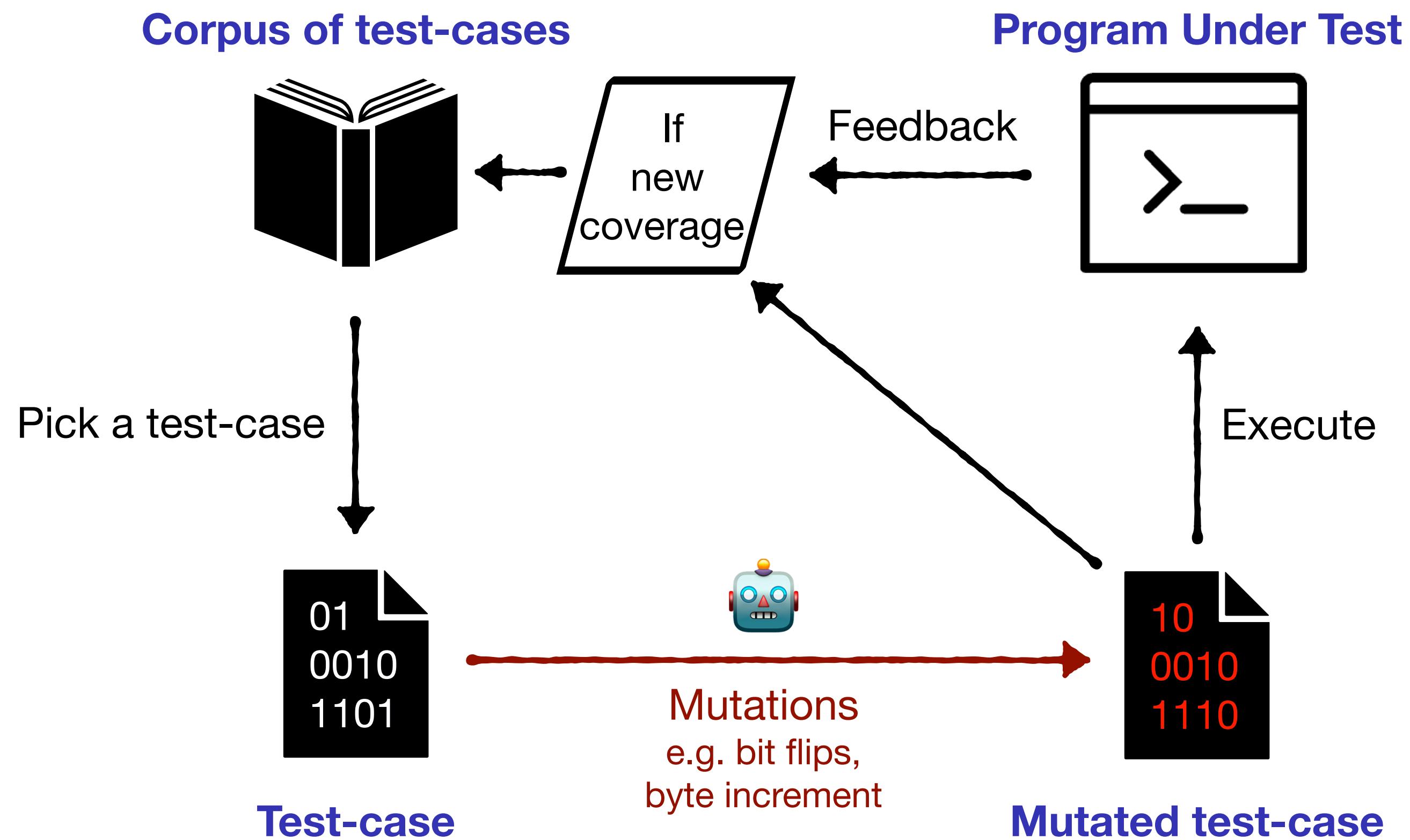
Bit-level Fuzzing (AFL-like)



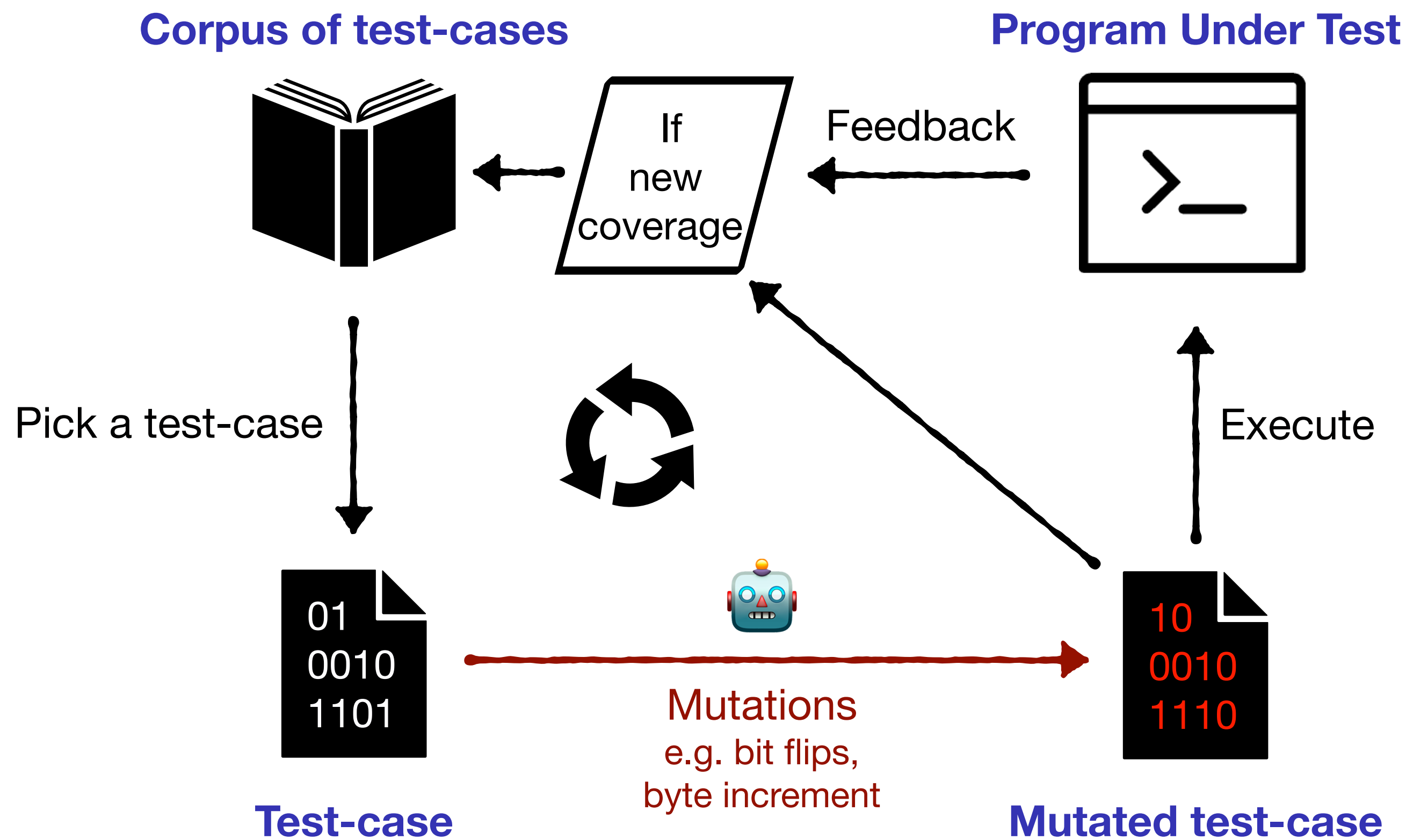
Bit-level Fuzzing (AFL-like)



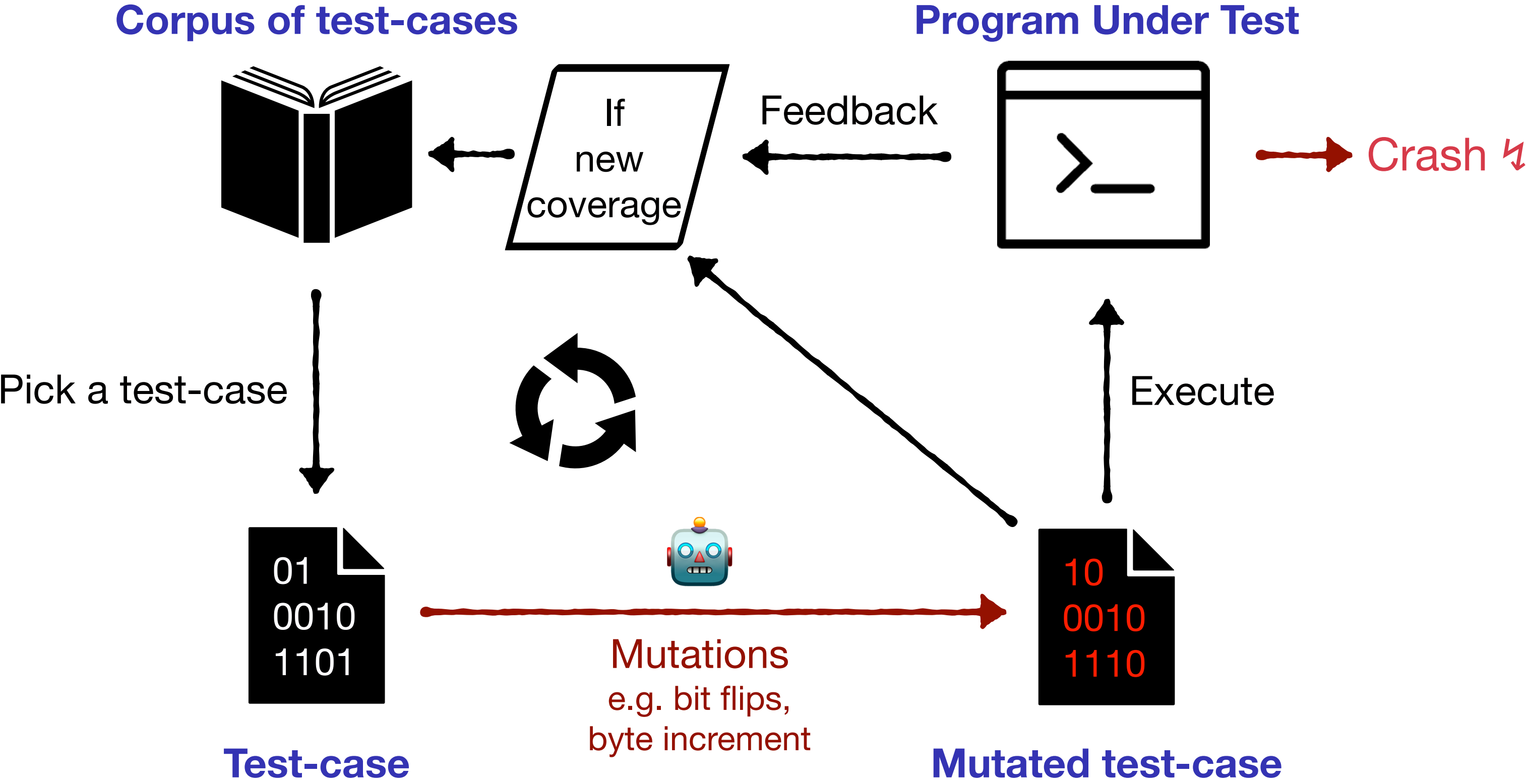
Bit-level Fuzzing (AFL-like)



Bit-level Fuzzing (AFL-like)



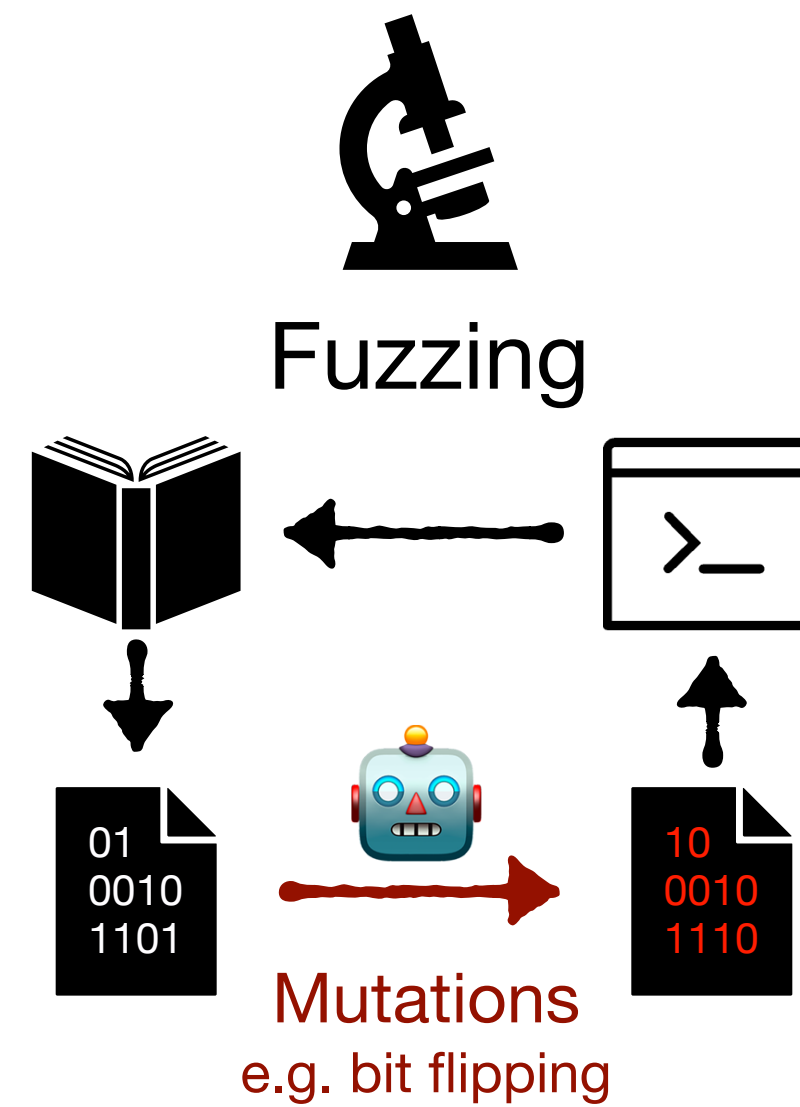
Bit-level Fuzzing (AFL-like)



Bit-level Fuzzing (AFL-like)

```
int FuzzerTestOneInput(const uint8_t *buf, size_t len)
{
    server = SSL_new();
    SSL_set_bio(server);
    IO_write(in, buf, len);

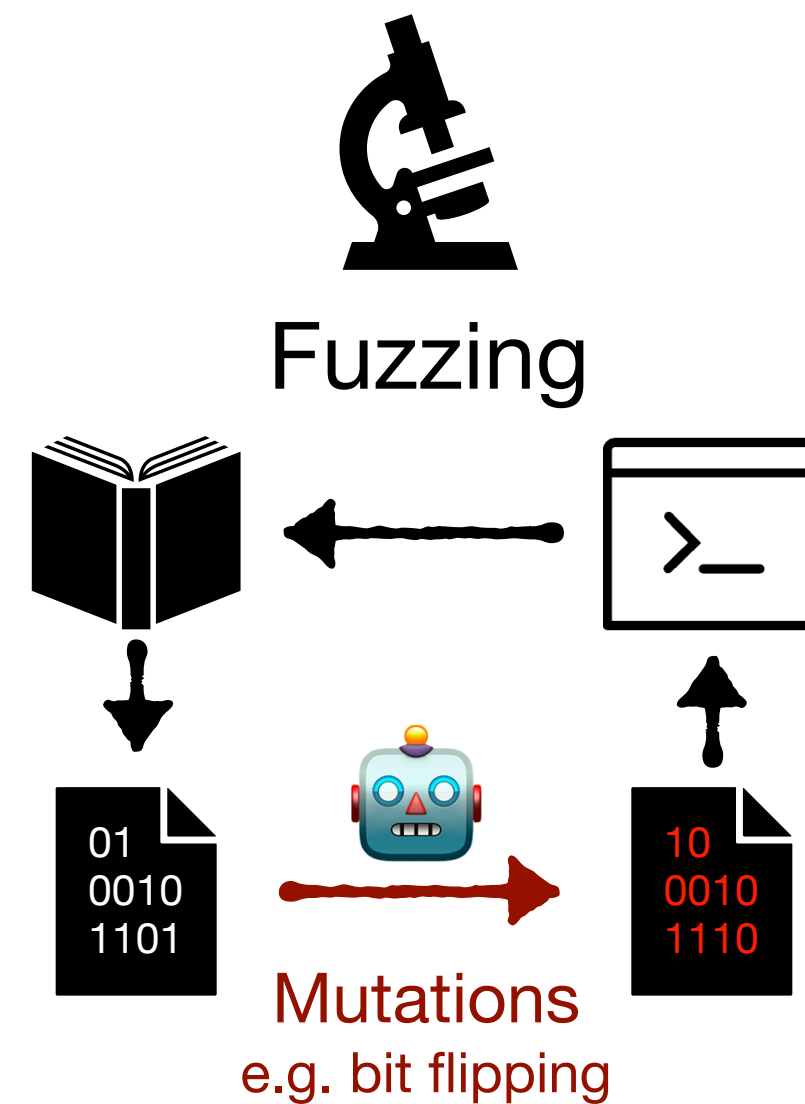
    if (SSL_do_handshake(server) == 1) {
        /* Keep reading data */
        uint8_t tmp[1024];
        for (;;) {
            if (SSL_read(server, tmp) <= 0) {
                break;
            }
        }
    }
    return 0;
}
```



Bit-level Fuzzing (AFL-like)

```
int FuzzerTestOneInput(const uint8_t *buf, size_t len)
{
    server = SSL_new();
    SSL_set_bio(server);
    IO_write(in, buf, len);

    if (SSL_do_handshake(server) == 1) {
        /* Keep reading data */
        uint8_t tmp[1024];
        for (;;) {
            if (SSL_read(server, tmp) <= 0) {
                break;
            }
        }
    }
    return 0;
}
```

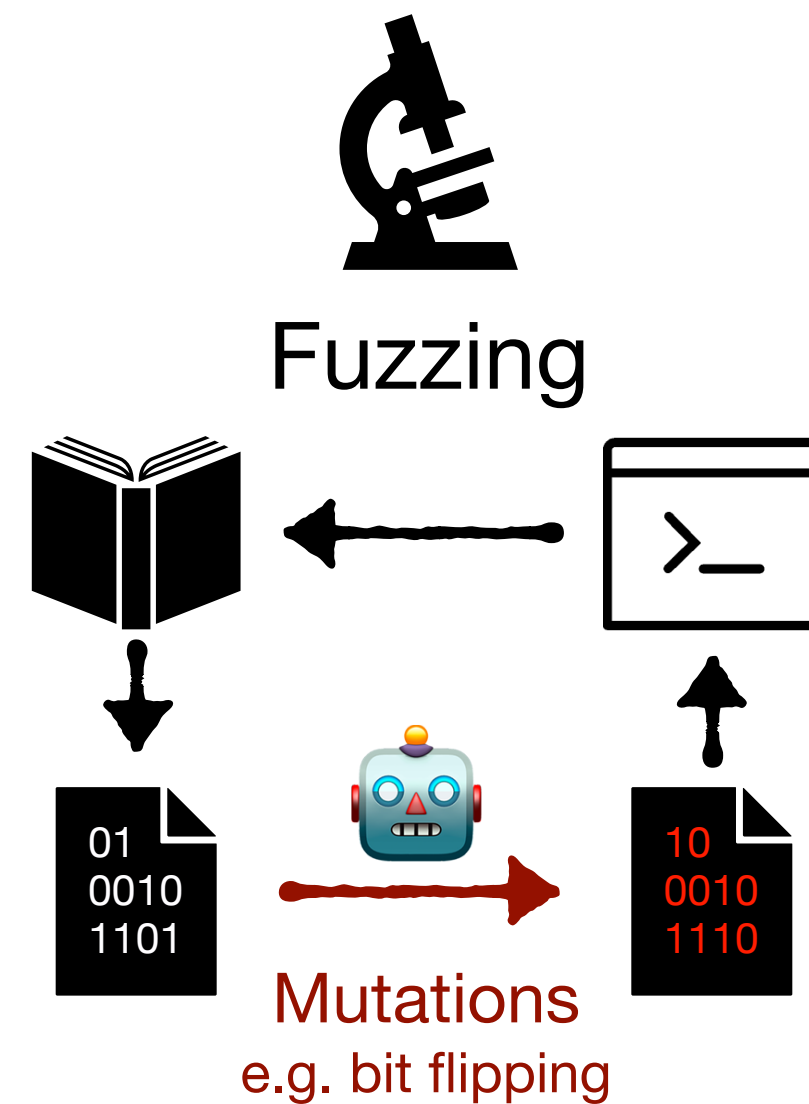


As for any cryptographic protocol, TLS uses several cryptographic primitives
e.g. ECDSA signature over full transcript

Bit-level Fuzzing (AFL-like)

```
int FuzzerTestOneInput(const uint8_t *buf, size_t len)
{
    server = SSL_new();
    SSL_set_bio(server);
    IO_write(in, buf, len);

    if (SSL_do_handshake(server) == 1) {
        /* Keep reading data */
        uint8_t tmp[1024];
        for (;;) {
            if (SSL_read(server, tmp) <= 0) {
                break;
            }
        }
    }
    return 0;
}
```



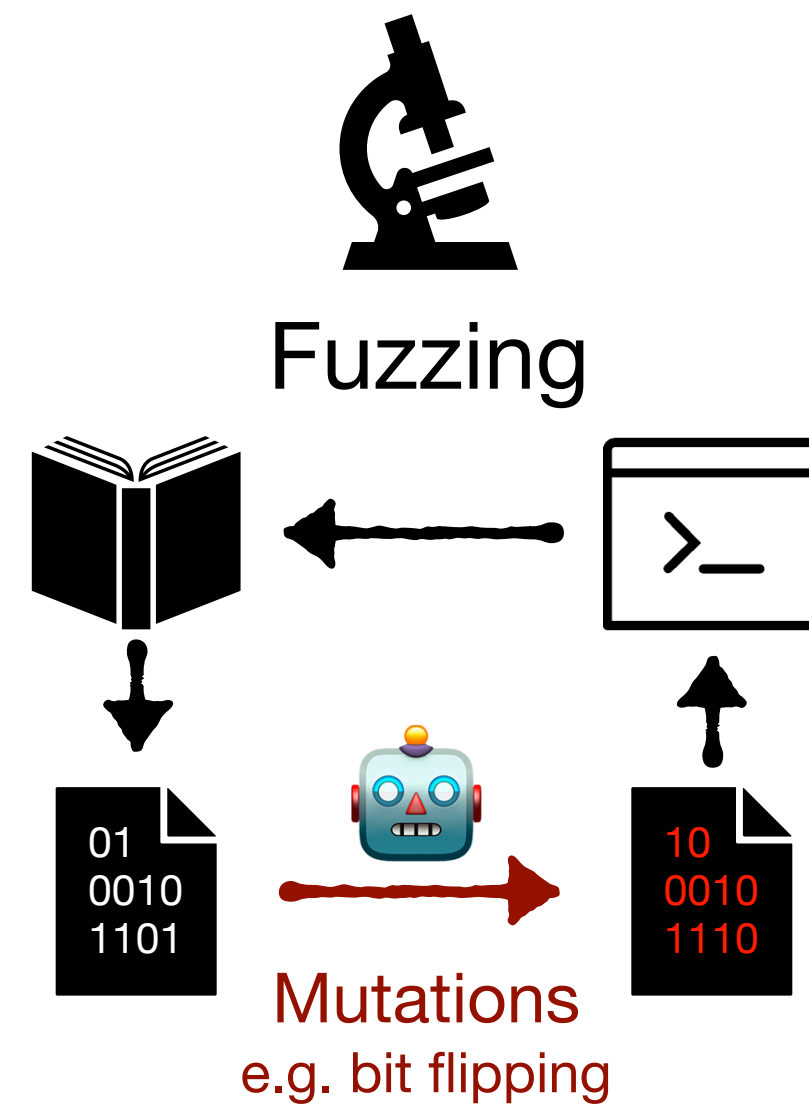
As for any cryptographic protocol, TLS uses several cryptographic primitives
e.g. ECDSA signature over full transcript

$\Pr[\text{Bitstring-level mutations} \text{ modify handshake message and adapt the ECDSA signature}]$
 $\leq \Pr[\text{break ECDSA}]$

Bit-level Fuzzing (AFL-like)

```
int FuzzerTestOneInput(const uint8_t *buf, size_t len)
{
    server = SSL_new();
    SSL_set_bio(server);
    IO_write(in, buf, len);

    if (SSL_do_handshake(server) == 1) {
        /* Keep reading data */
        uint8_t tmp[1024];
        for (;;) {
            if (SSL_read(server, tmp) <= 0) {
                break;
            }
        }
    }
    return 0;
}
```



As for any cryptographic protocol, TLS uses several cryptographic primitives e.g. ECDSA signature over full transcript

$\Pr[\text{Bitstring-level mutations} \text{ modify handshake message and adapt the ECDSA signature}] \leq \Pr[\text{break ECDSA}]$

✗ Fuzzing Limitation 1:
Bitstring-level mutations only

No structural message/message flow modification e.g. negligible probability of computing cryptographic and other structural modifications through bit-level mutations

👉 Reachability problem

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

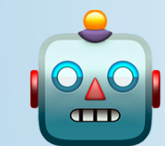
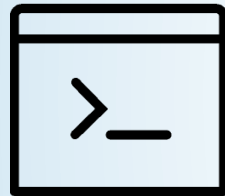
- State-machine fuzzers/learners where:
 - **Input state** = state-machine model: e.g. finite alphabet, automaton
 - **Test-case** = series of actions (from alphabet), e.g. ClientHello, ServerHello
- + Captures protocol vulnerabilities corresponding to state-machine violations in implementations, e.g. [22-25640] SkipVerify (auth. bypass with CertInfo but wo/ Cert)
- Reachability: no (structural) message modifications (except finite built-in mods)
 - 👉 does not fully capture the class of logical attacks
- Detection: manual/incomplete vulnerability detections (memory/protocol vulns)

Memory Safety Vulnerabilities

⚡ HeartBleed



Fuzzing



01
0010
1101

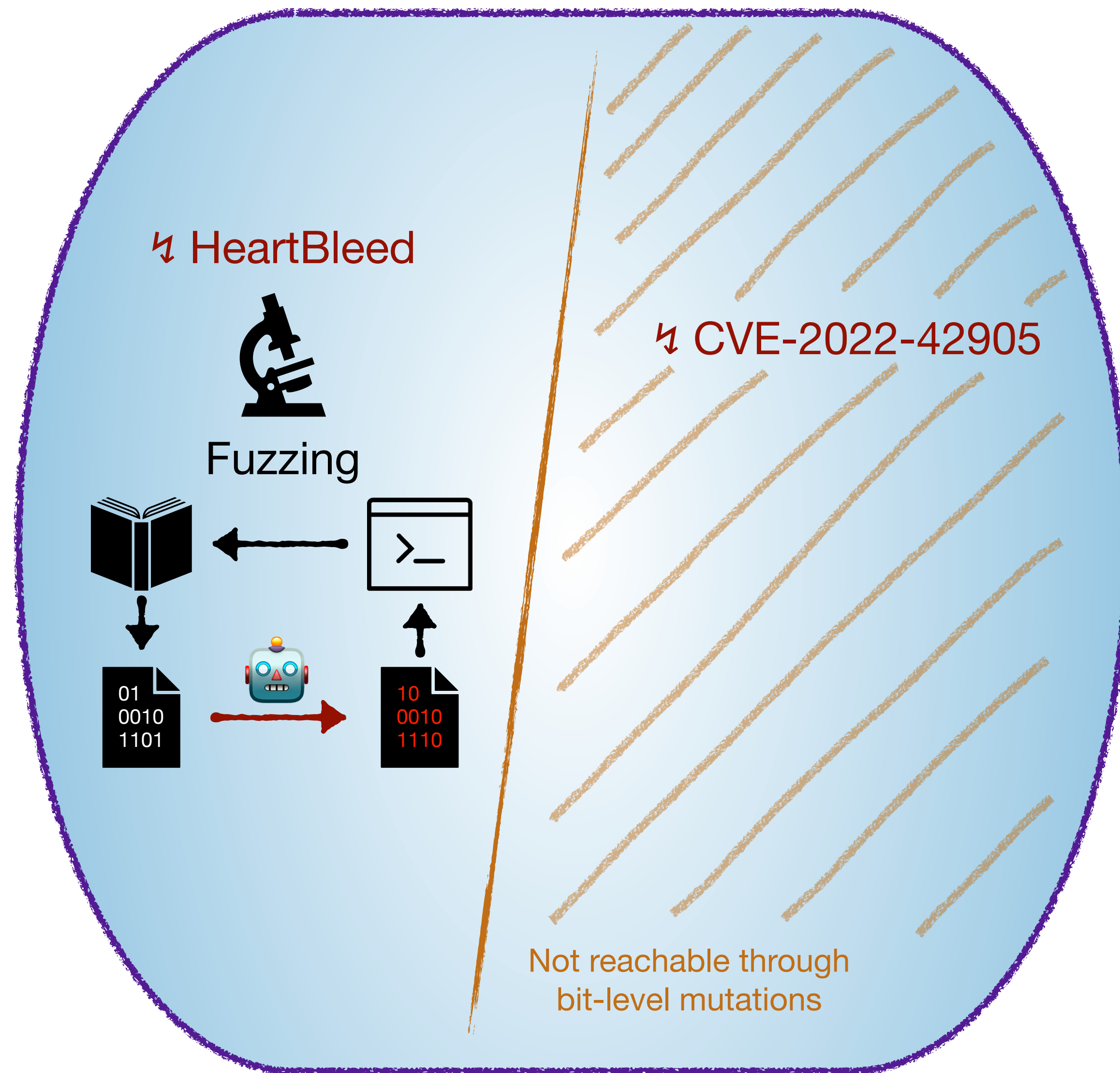
10
0010
1110

⚡ CVE-2022-42905

Not reachable through
bit-level mutations

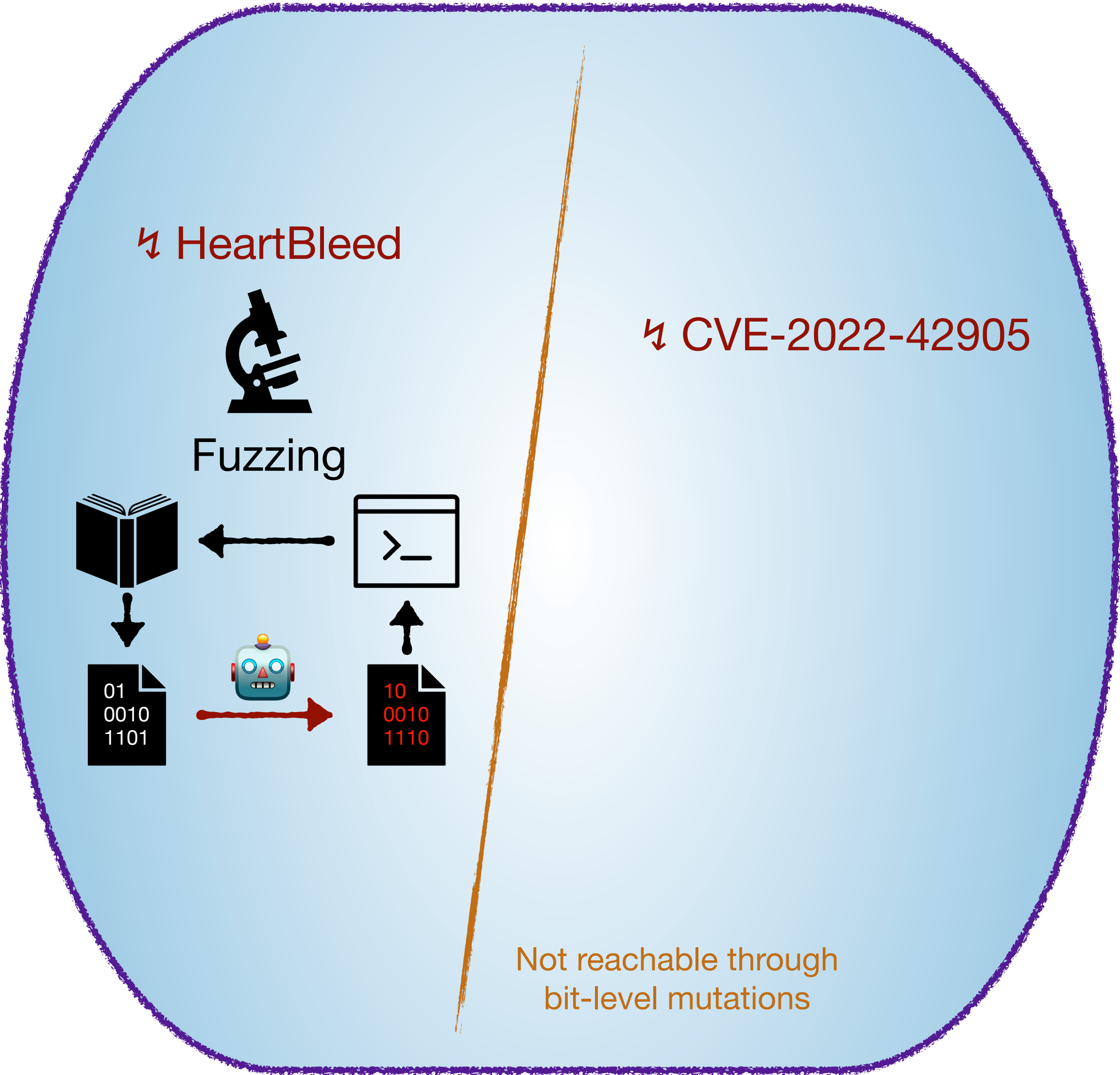
(e.g. buffer-overflow)

Memory Safety Vulnerabilities



(e.g. buffer-overflow)

Memory Safety Vulnerabilities



(e.g. buffer-overflow)

Protocol Vulnerabilities



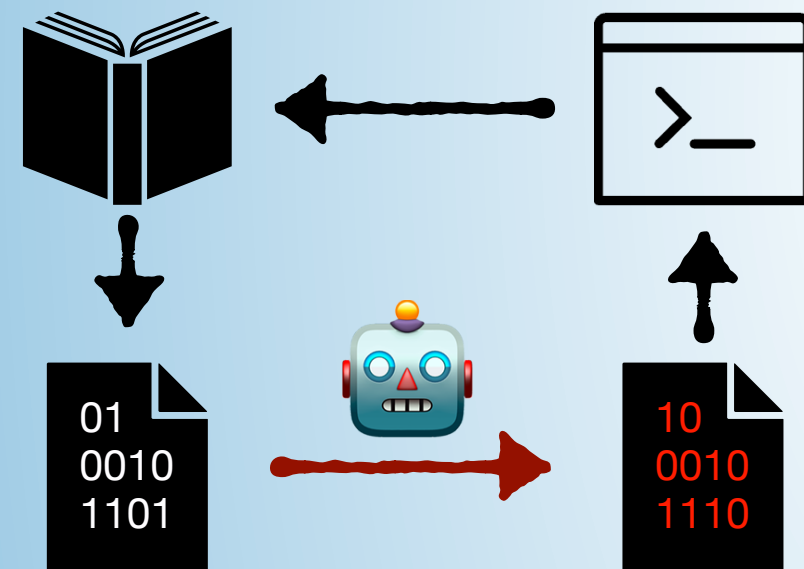
(e.g. authentication bypass)

Memory Safety Vulnerabilities

⚡ HeartBleed



Fuzzing



⚡ CVE-2022-42905

Not reachable through
bit-level mutations

Protocol Vulnerabilities

⚡ goto fail;

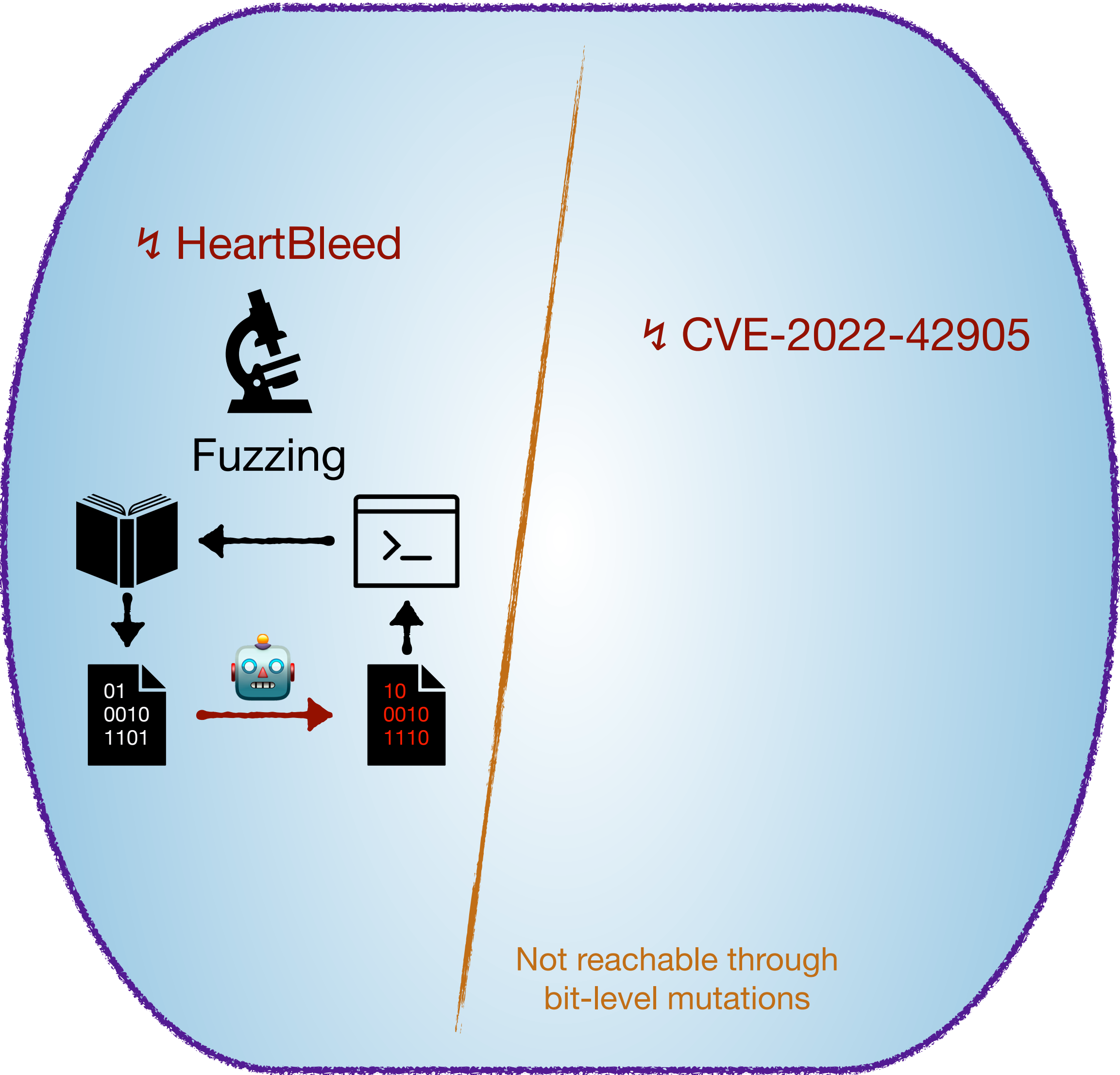
⚡ 3HandShake

(no crash here)

Not detected
Not reachable through
bit-level mutations

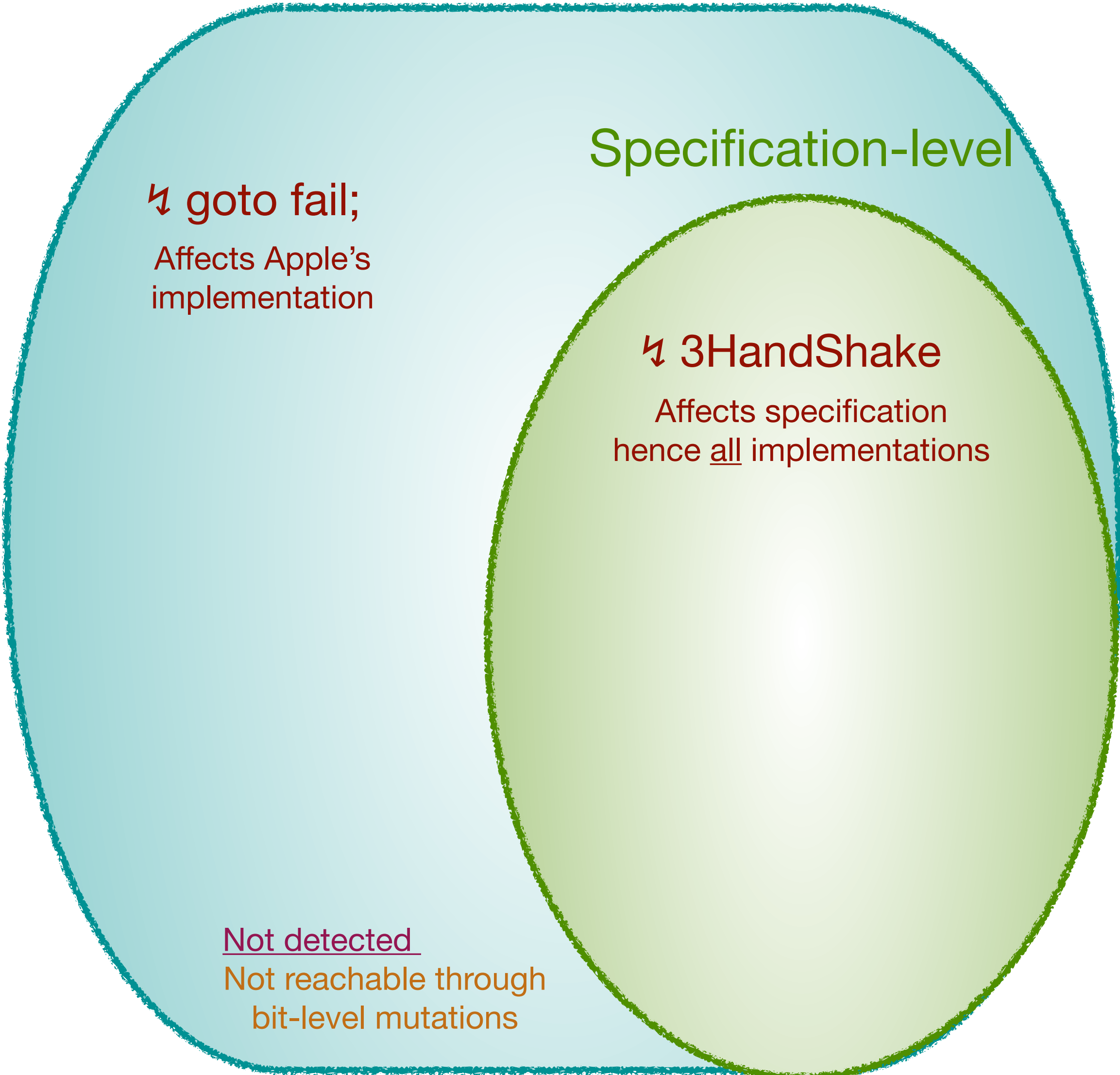
✗ Fuzzing Limitation 2: inability to detect protocol vulnerabilities ➡ Detection problem

Memory Safety Vulnerabilities



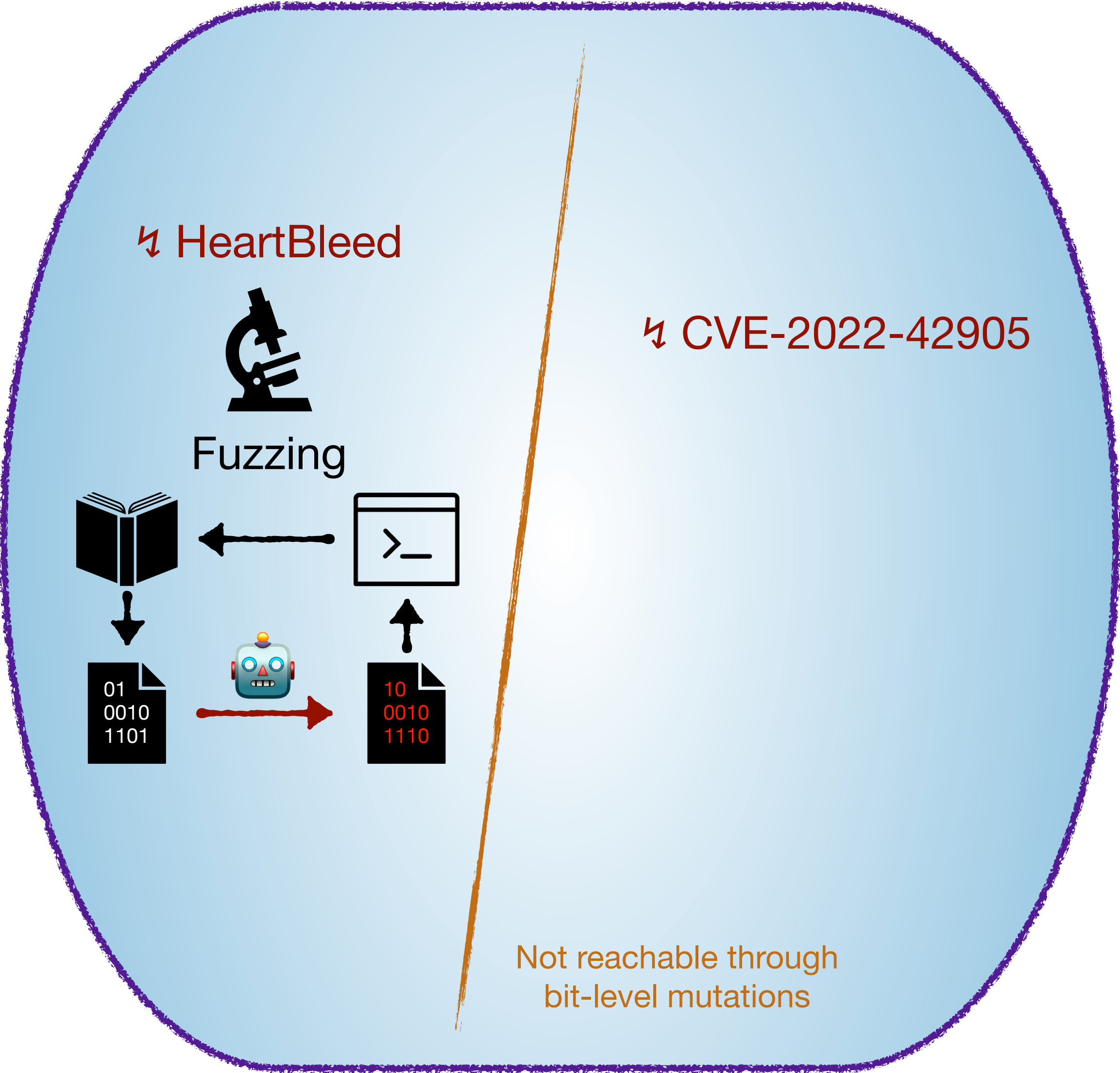
(e.g. buffer-overflow)

Protocol Vulnerabilities



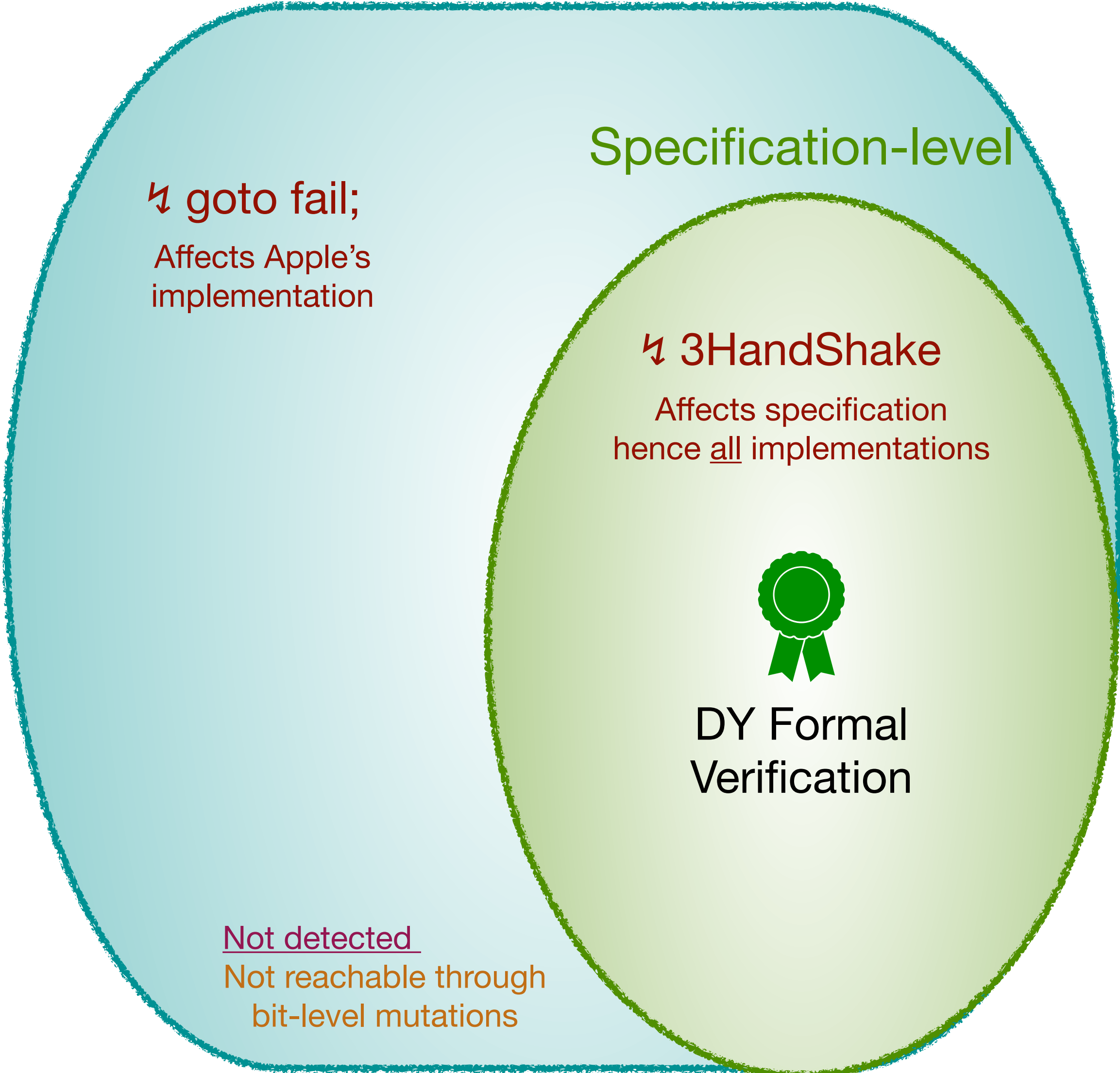
(e.g. authentication bypass)

Memory Safety Vulnerabilities



(e.g. buffer-overflow)

Protocol Vulnerabilities



(e.g. authentication bypass)

Dolev-Yao Verification 🏆

Limitations

Dolev-Yao Formal Model and Verification

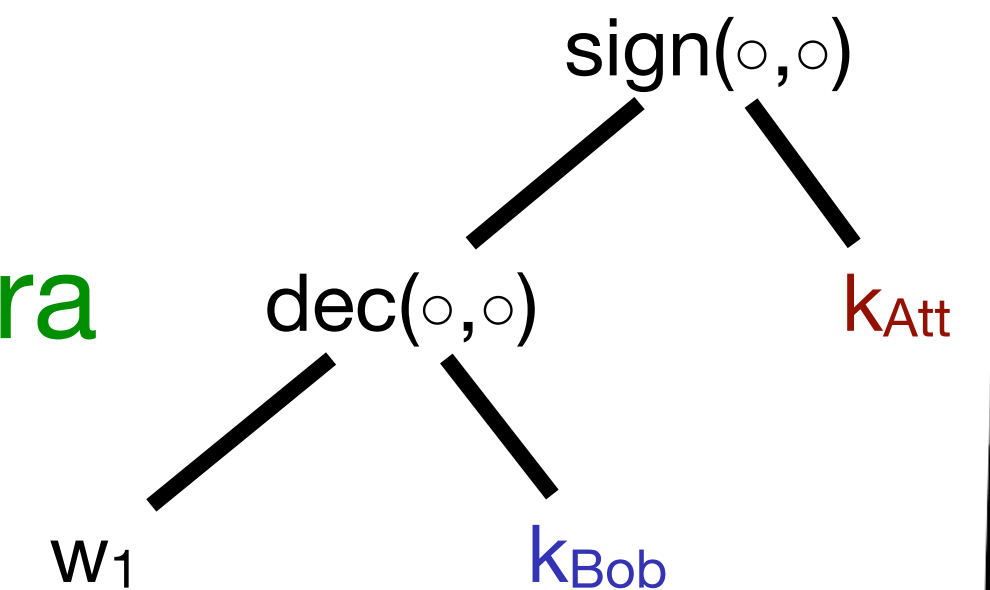


👉 Mathematical model for analyzing cryptographic protocols and threat models

Threat model 🤡:

- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

« Messages as formal terms » paradigm: **messages model = term algebra**
e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$



Dolev-Yao Formal Model and Verification

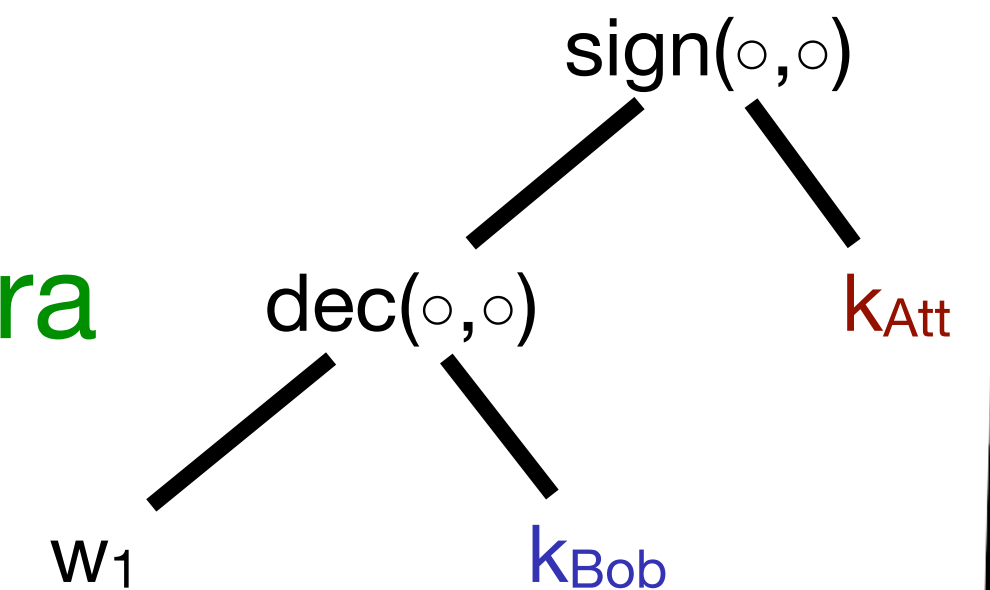


👉 Mathematical model for analyzing cryptographic protocols and threat models

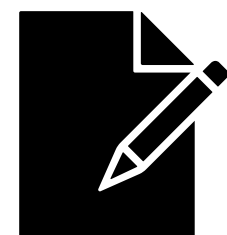
Threat model 🤡:

- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

« Messages as formal terms » paradigm: **messages model = term algebra**
e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$



Protocol
specification



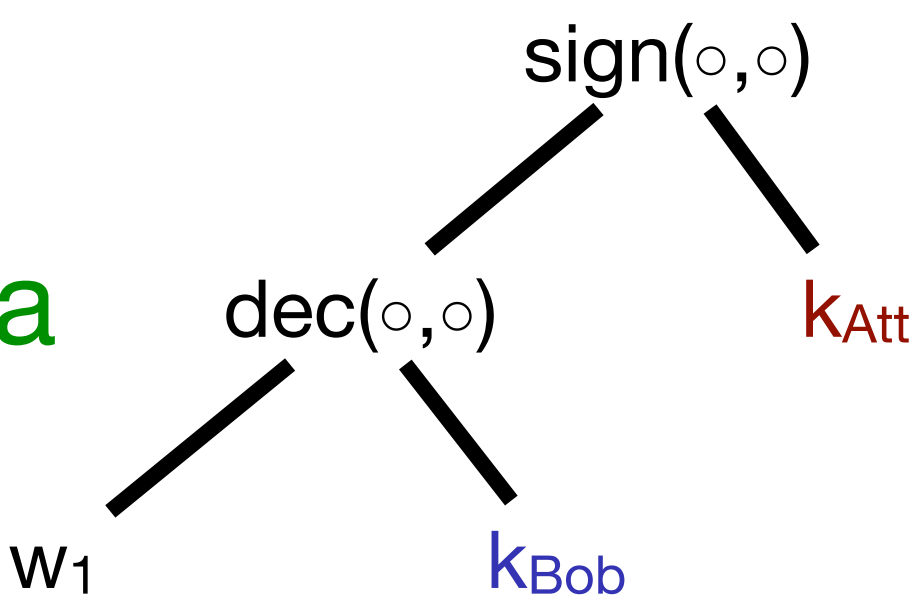
Dolev-Yao Formal Model and Verification

👉 Mathematical model for analyzing cryptographic protocols and threat models

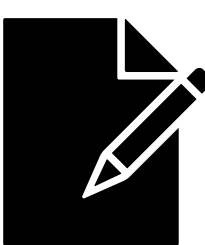
Threat model :

- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

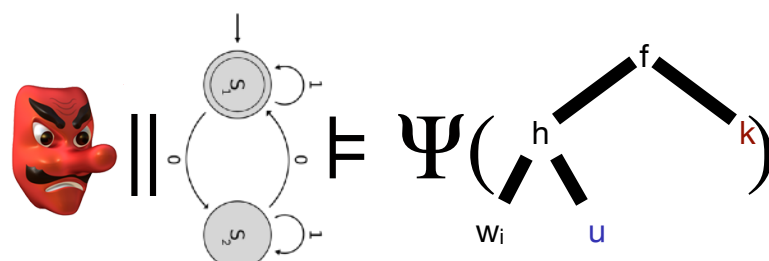
« Messages as formal terms » paradigm: **messages model = term algebra**
e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$



Protocol
specification



Modeling



Formal model

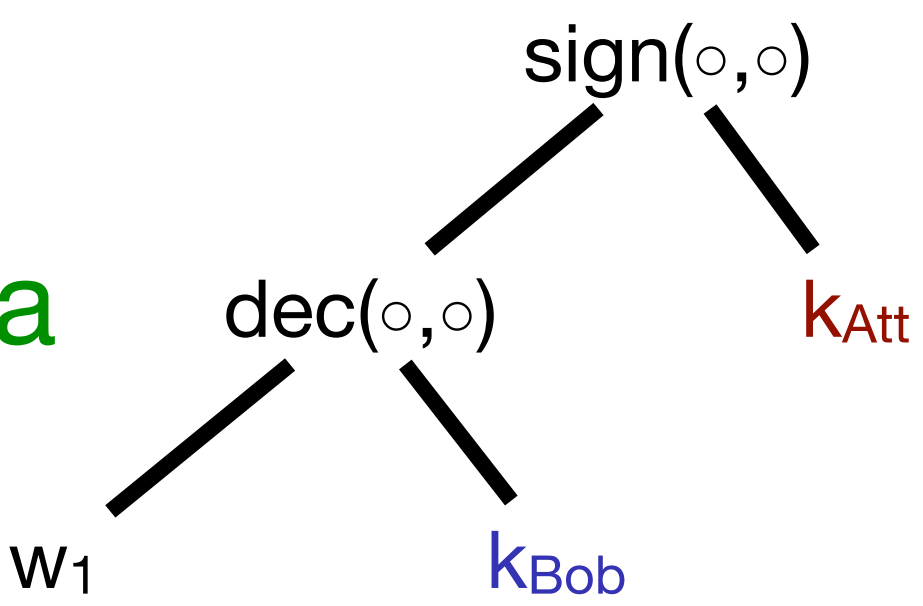
Dolev-Yao Formal Model and Verification

👉 Mathematical model for analyzing cryptographic protocols and threat models

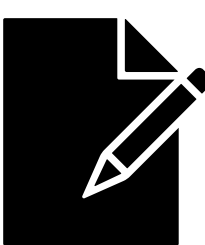
Threat model :

- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

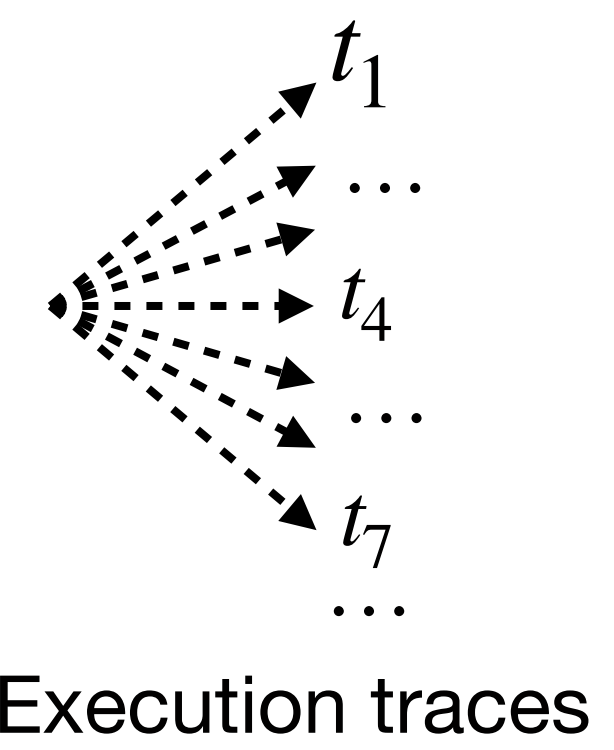
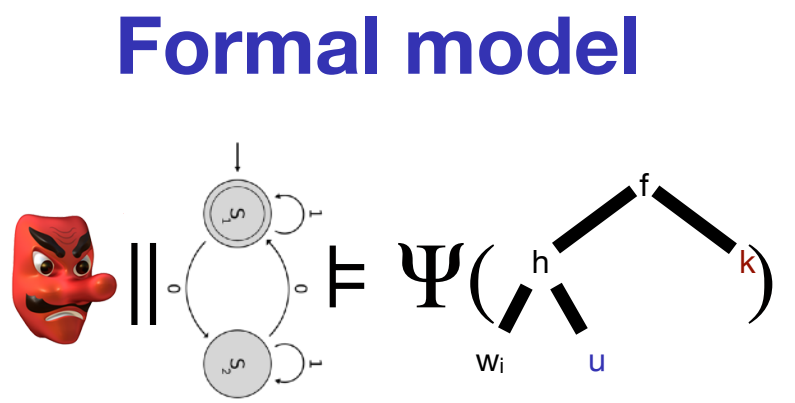
« Messages as formal terms » paradigm: **messages model = term algebra**
e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$



Protocol
specification



Modeling



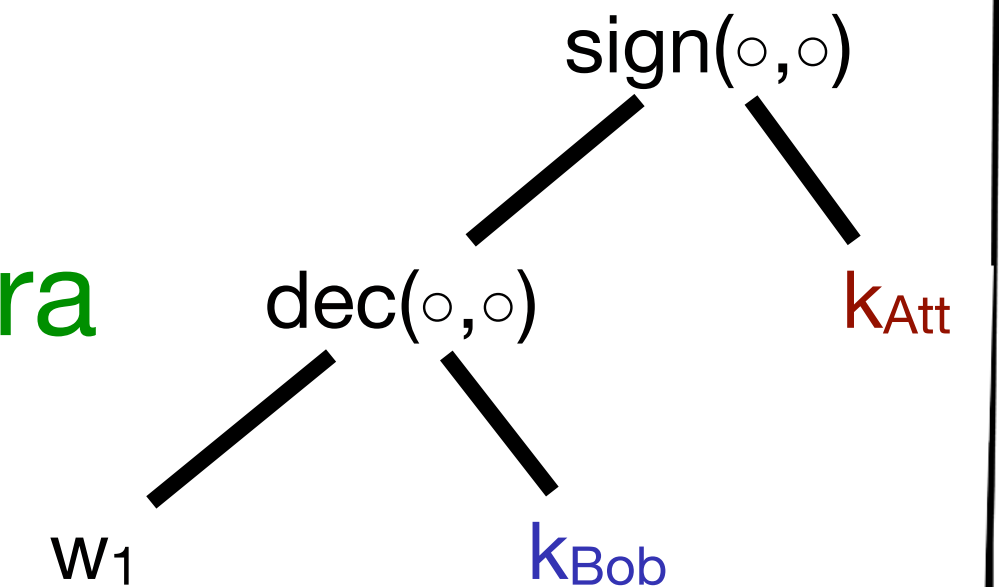


- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

« Messages as formal terms »

paradigm: messages model = term algebra

e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$

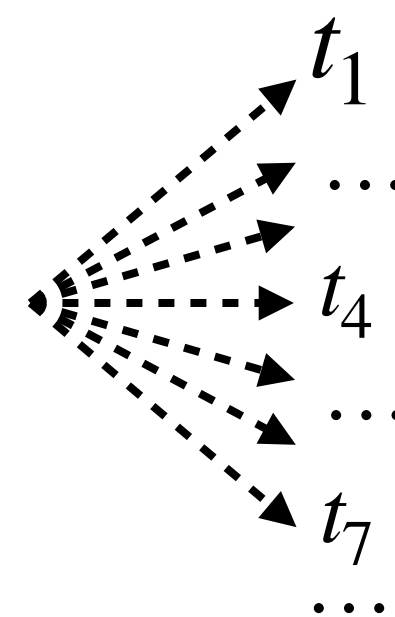
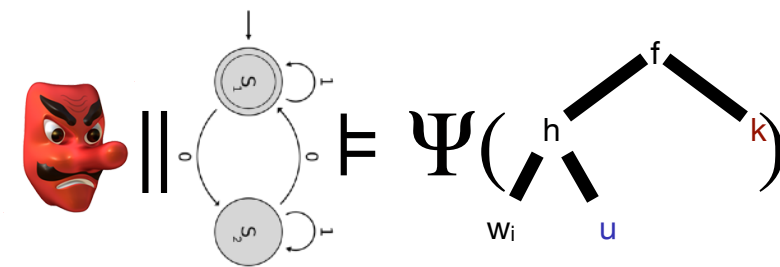


Protocol specification

Formal model



Modeling



Execution traces

$$t_1 = \mathbf{out}(\text{client}, w_1).$$

in(serv, w₁). // attacker  only relays message w₁ to serv

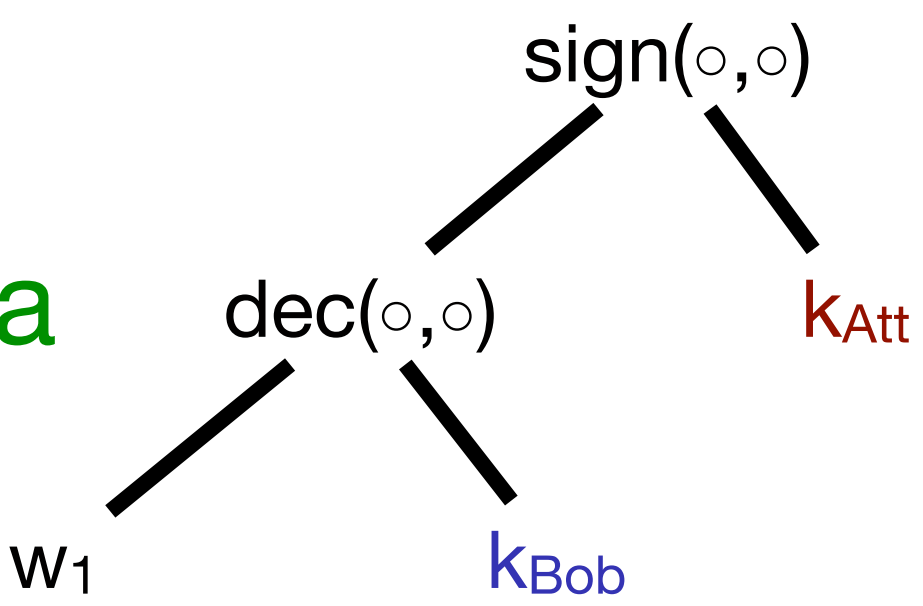
Dolev-Yao Formal Model and Verification

👉 Mathematical model for analyzing cryptographic protocols and threat models

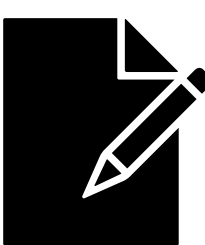
Threat model :

- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

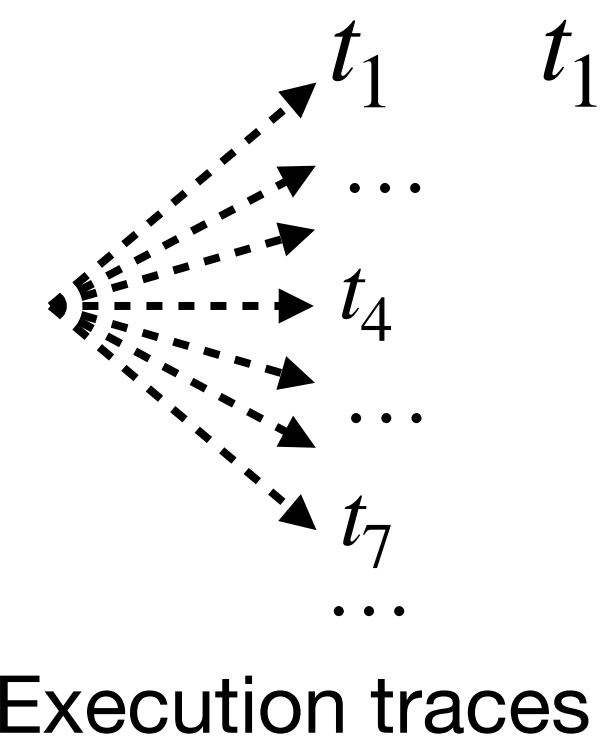
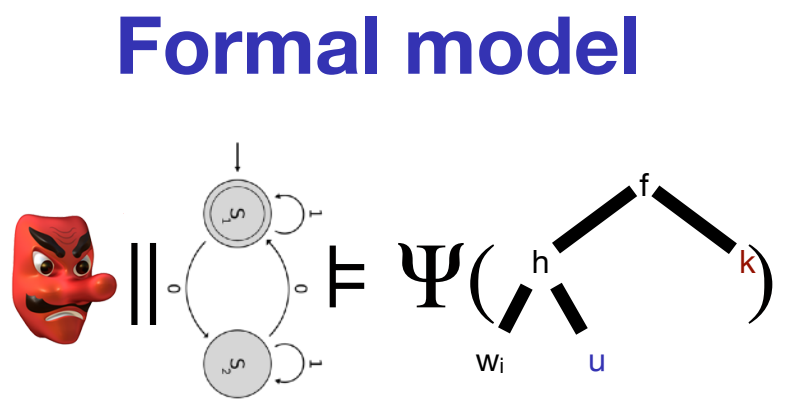
« Messages as formal terms » paradigm: **messages model = term algebra**
e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$



Protocol
specification



Modeling



$t_1 = \mathbf{out}(\text{client}, w_1).$
 $\mathbf{in}(\text{serv}, w_1).$ // attacker  only relays message w_1 to serv
 $\mathbf{out}(\text{serv}, w_2).$
 $\mathbf{in}(\text{client}, \text{sign}(\text{dec}(w_2, k_{\text{Bob}}), k_{\text{Att}}))$
// attacker  computes a new **term** out of w_2 and sends it to client

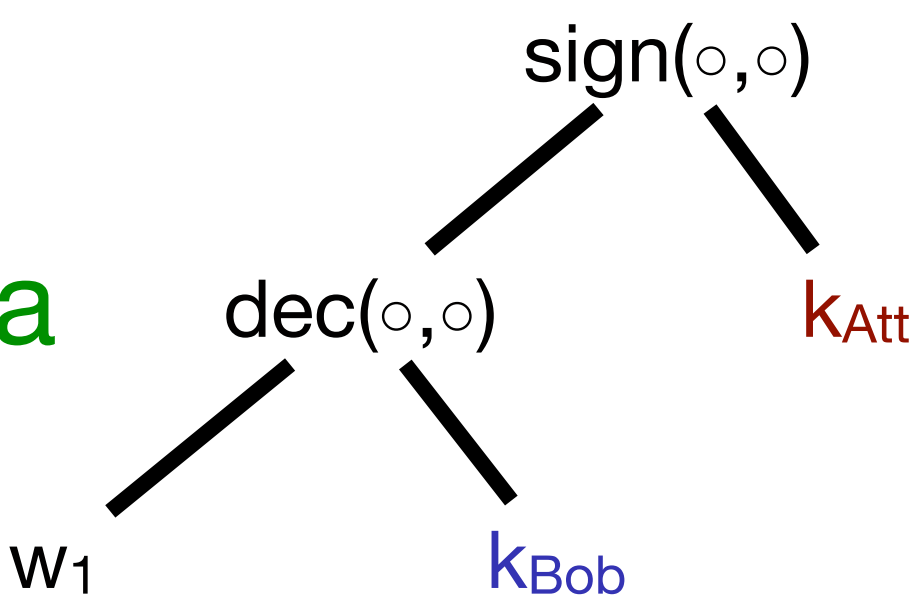
Dolev-Yao Formal Model and Verification

👉 Mathematical model for analyzing cryptographic protocols and threat models

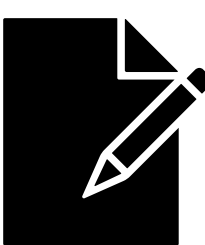
Threat model :

- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

« Messages as formal terms » paradigm: **messages model = term algebra**
e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$

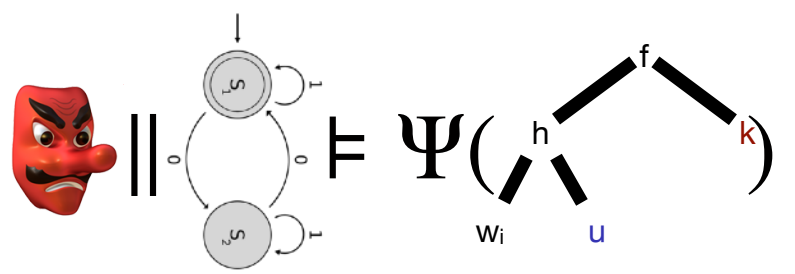


Protocol
specification



Modeling
→

Formal model



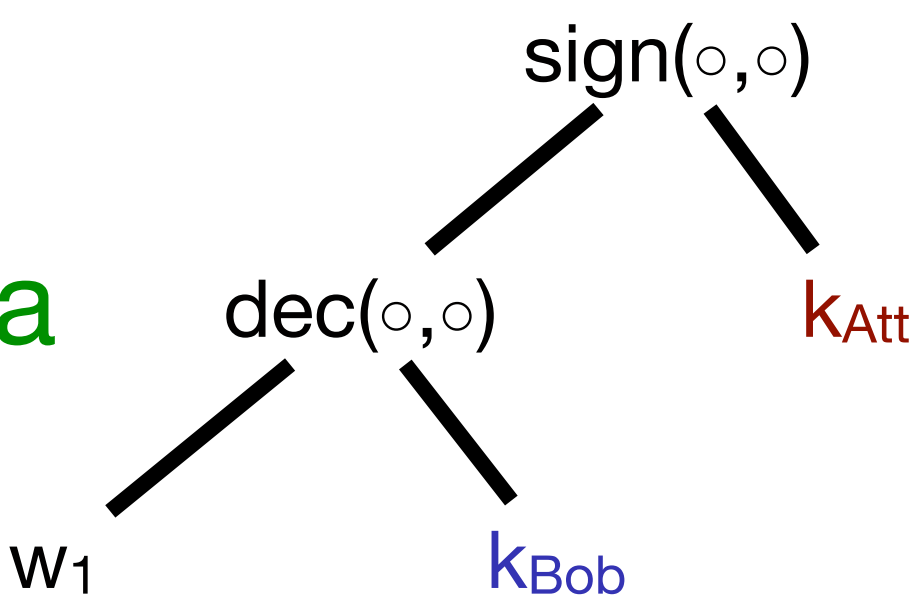
Dolev-Yao Formal Model and Verification

👉 Mathematical model for analyzing cryptographic protocols and threat models

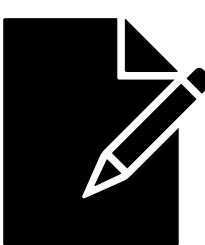
Threat model :

- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

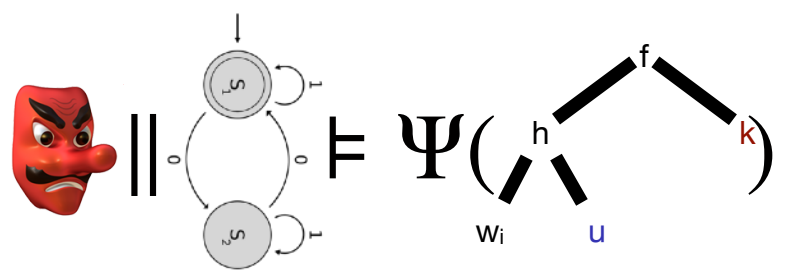
« Messages as formal terms » paradigm: **messages model = term algebra**
e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$



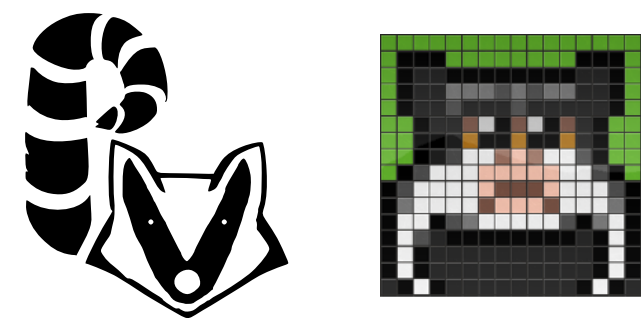
Protocol
specification



Modeling



Automated verification

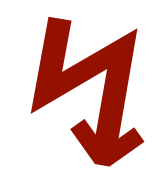


Tools: Proverif / Tamarin

Proofs



or



Attacks

Some "success stories": TLS 1.3, EMV, 5G, LAKE EDHOC, etc.

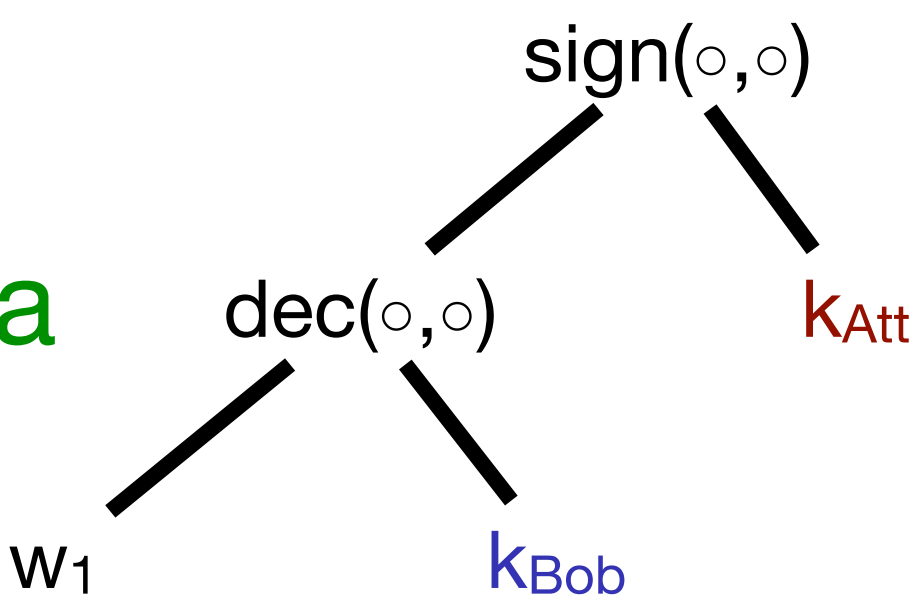
Dolev-Yao Formal Model and Verification

👉 Mathematical model for analyzing cryptographic protocols and threat models

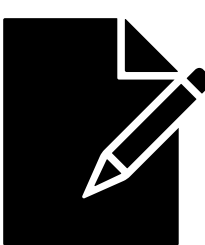
Threat model :

- active attacker controls the network: **intercept, modify, inject messages**
- is able to **use cryptography functionalities** (e.g. enc/dec)

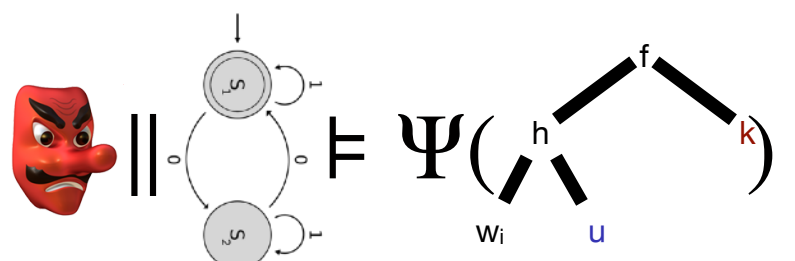
« Messages as formal terms » paradigm: **messages model = term algebra**
e.g. $\text{dec}(\circ, \circ)$, $\text{sign}(\circ, \circ)$



Protocol
specification



Modeling



Formal model

Automated verification

Proofs

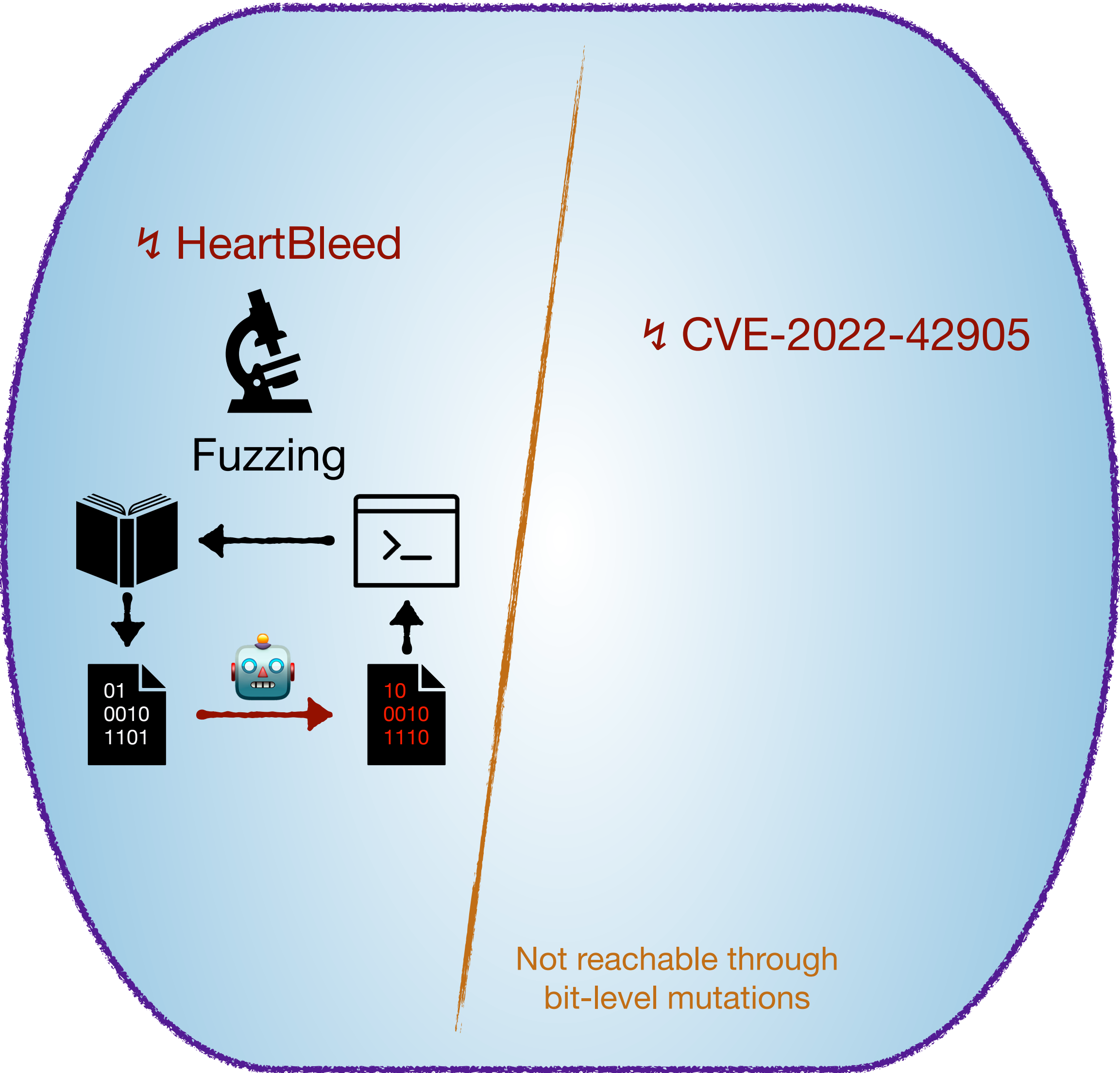


or

Some "success stories": TLS 1.3, EMV, 5G, LAKE EDHOC, etc.

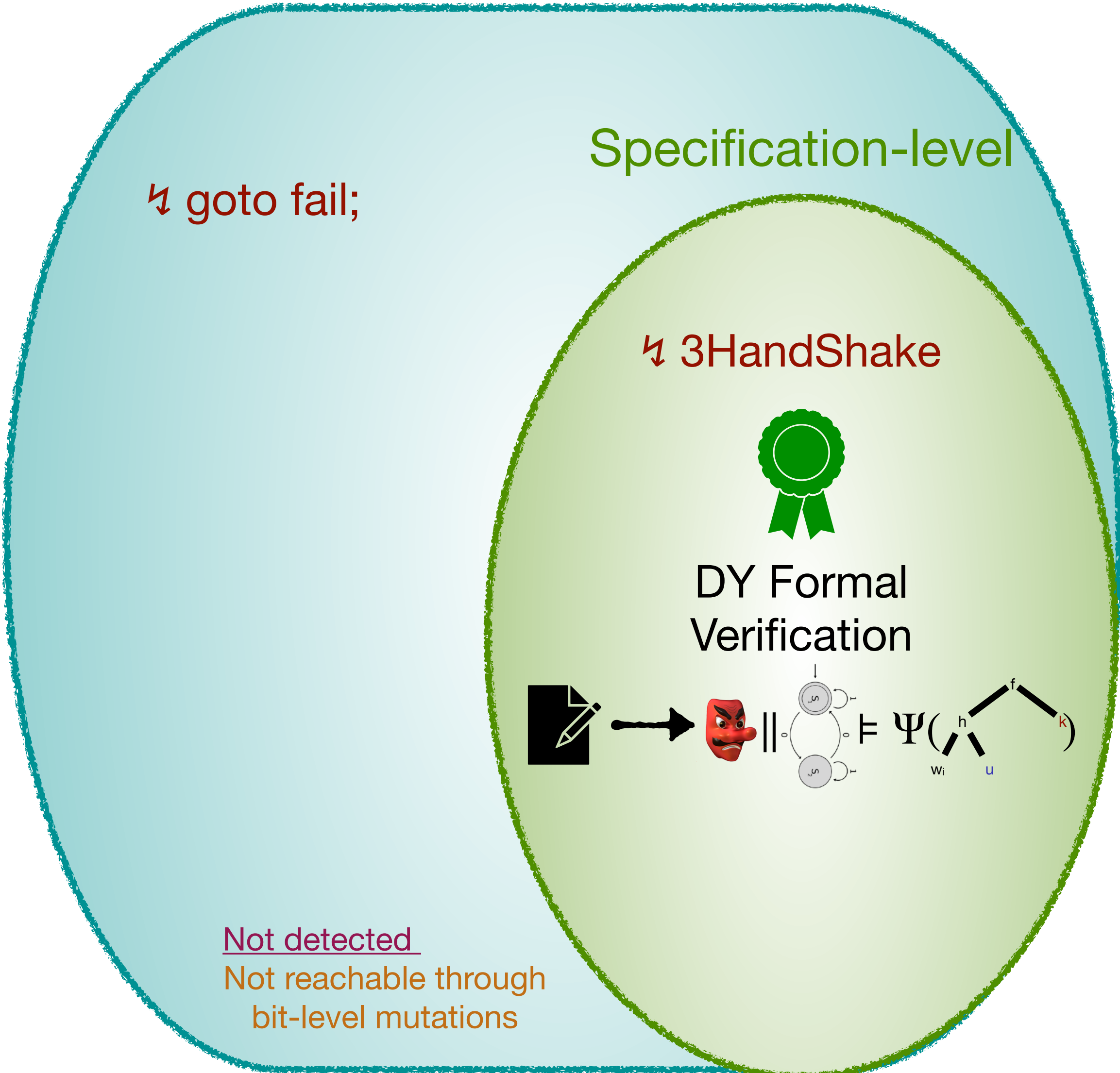
X DY Verification Limitation: No guarantee against implementation-level flaw

Memory Safety Vulnerabilities



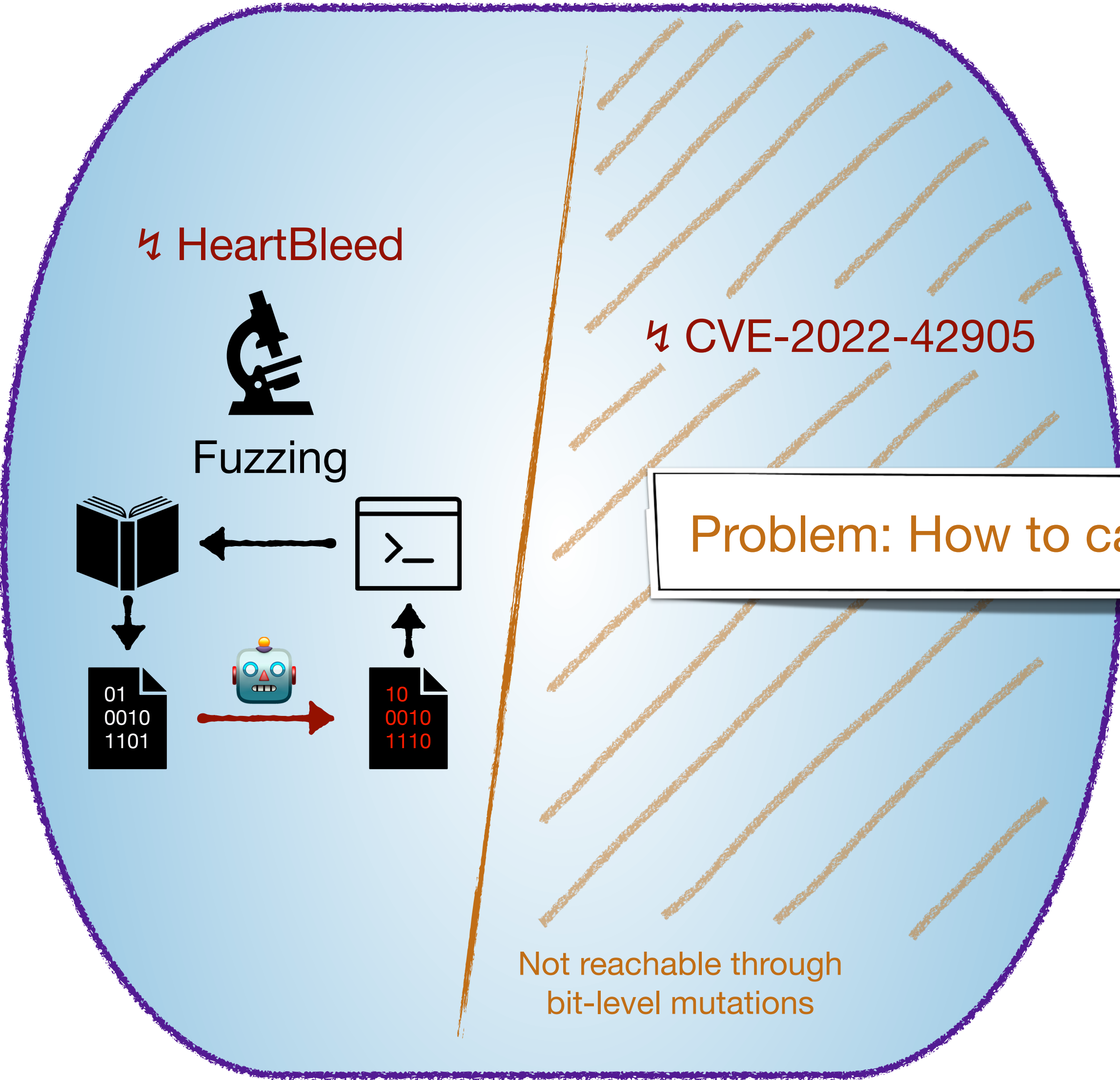
(e.g. buffer-overflow)

Protocol Vulnerabilities



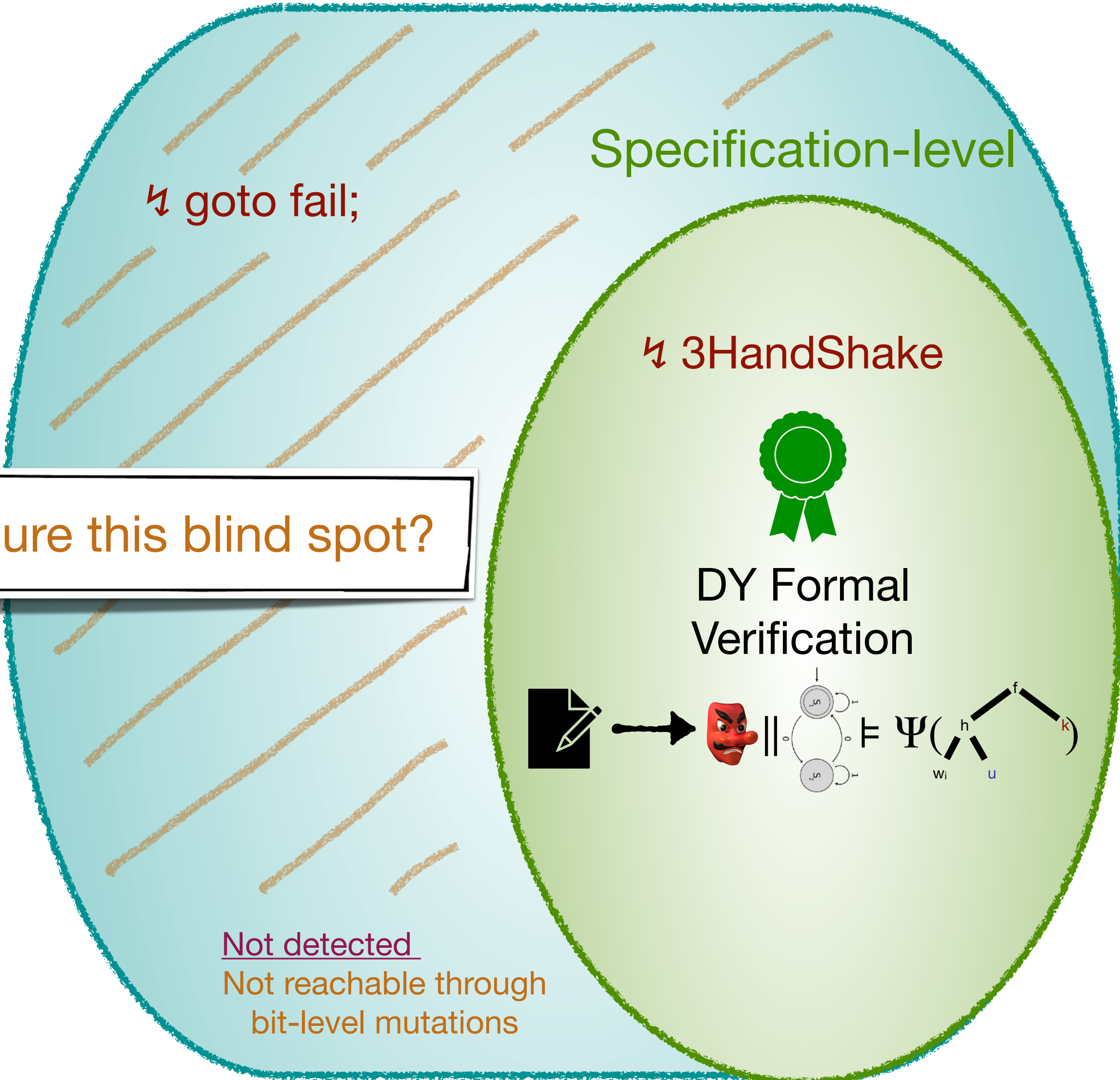
(e.g. authentication bypass)

Memory Safety Vulnerabilities



(e.g. buffer-overflow)

Protocol Vulnerabilities



(e.g. authentication bypass)

Problem: How to capture this blind spot?

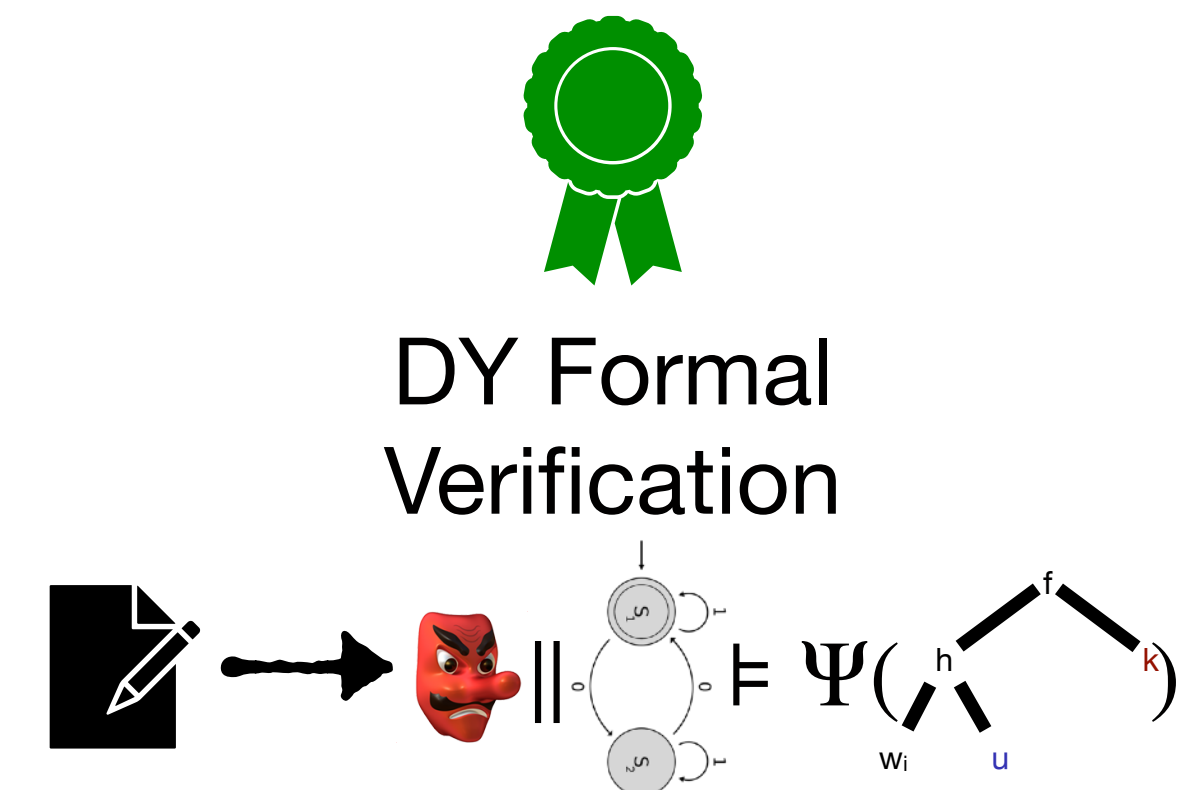
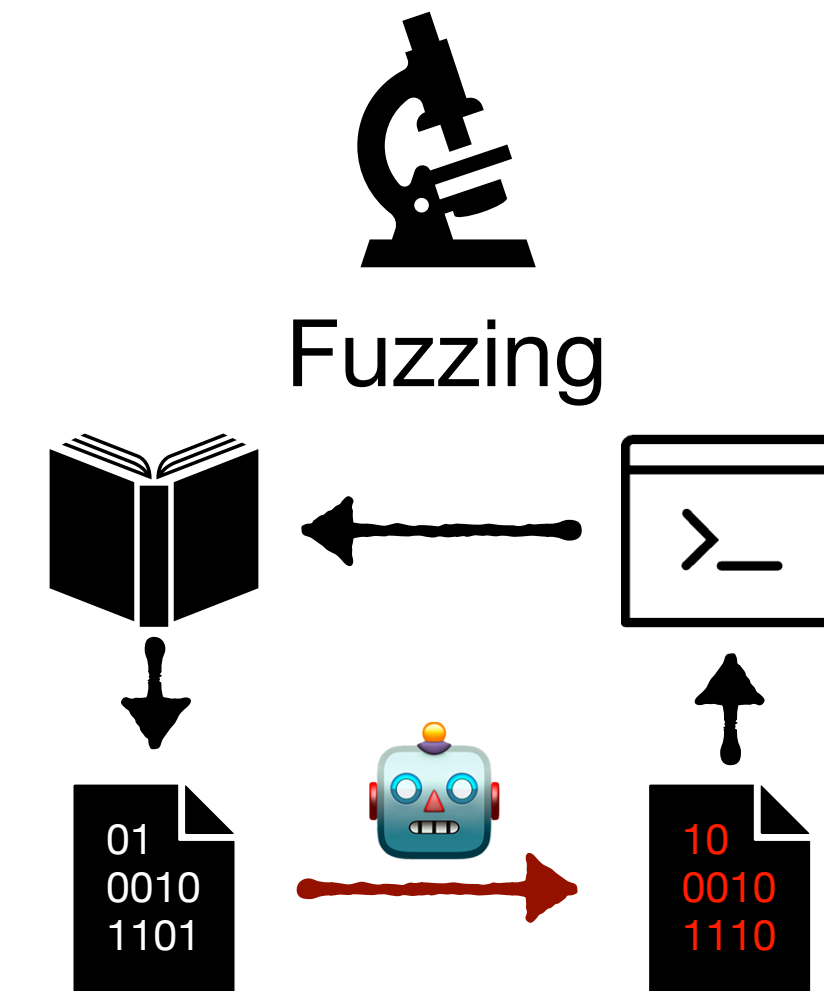
Securing Cryptographic Protocol Fuzz Implementations: The Blind Spot

✗ Fuzzing Limitation 1: Reachability
Bitstring-level mutations only

✗ Fuzzing Limitation 2: Detection
Inability to detect protocol vulnerabilities

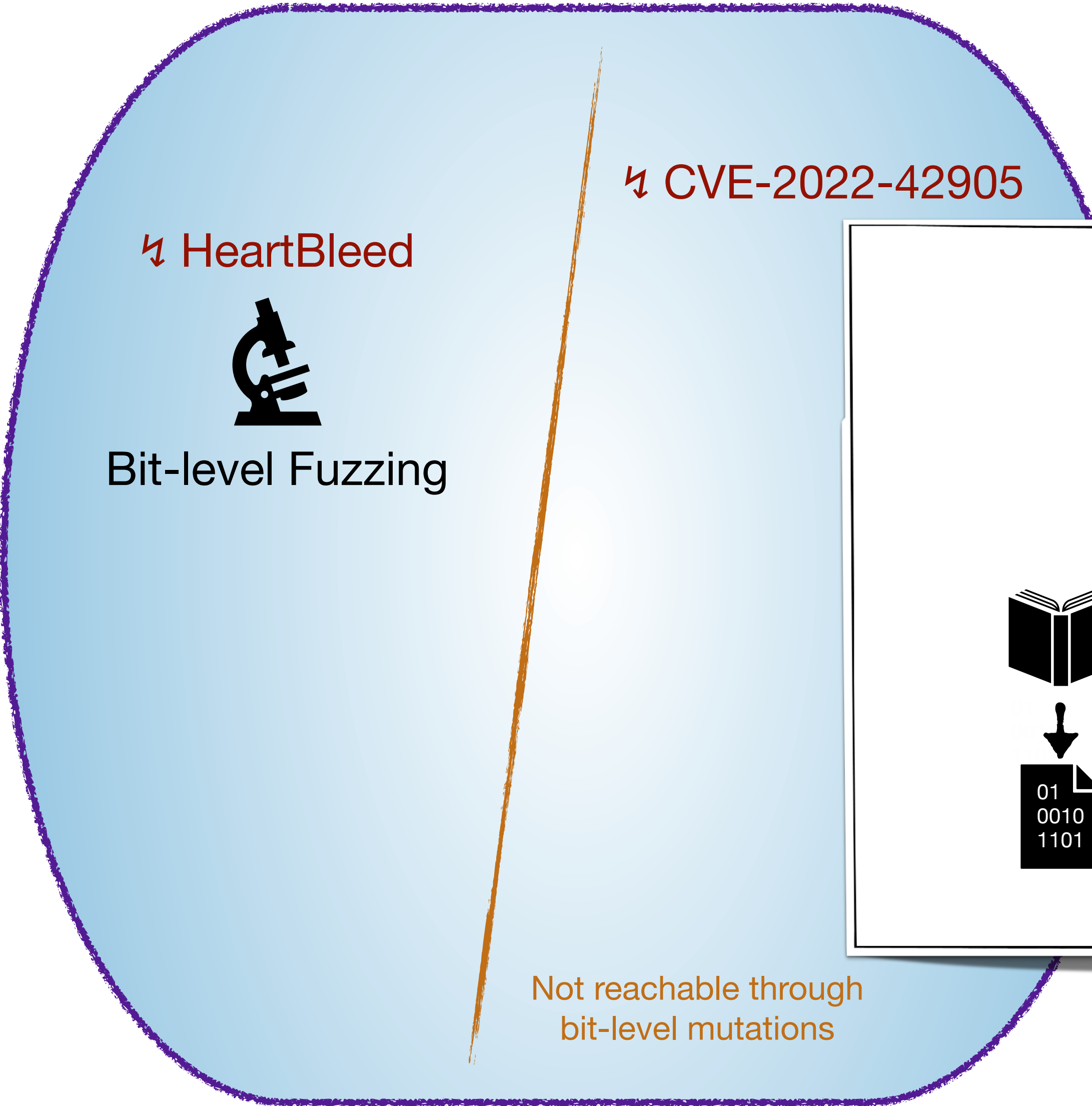
(similar limitations apply to State-Machine-based Fuzzers)

✗ DY Verification Limitation: Spec. only
No guarantee on implementation

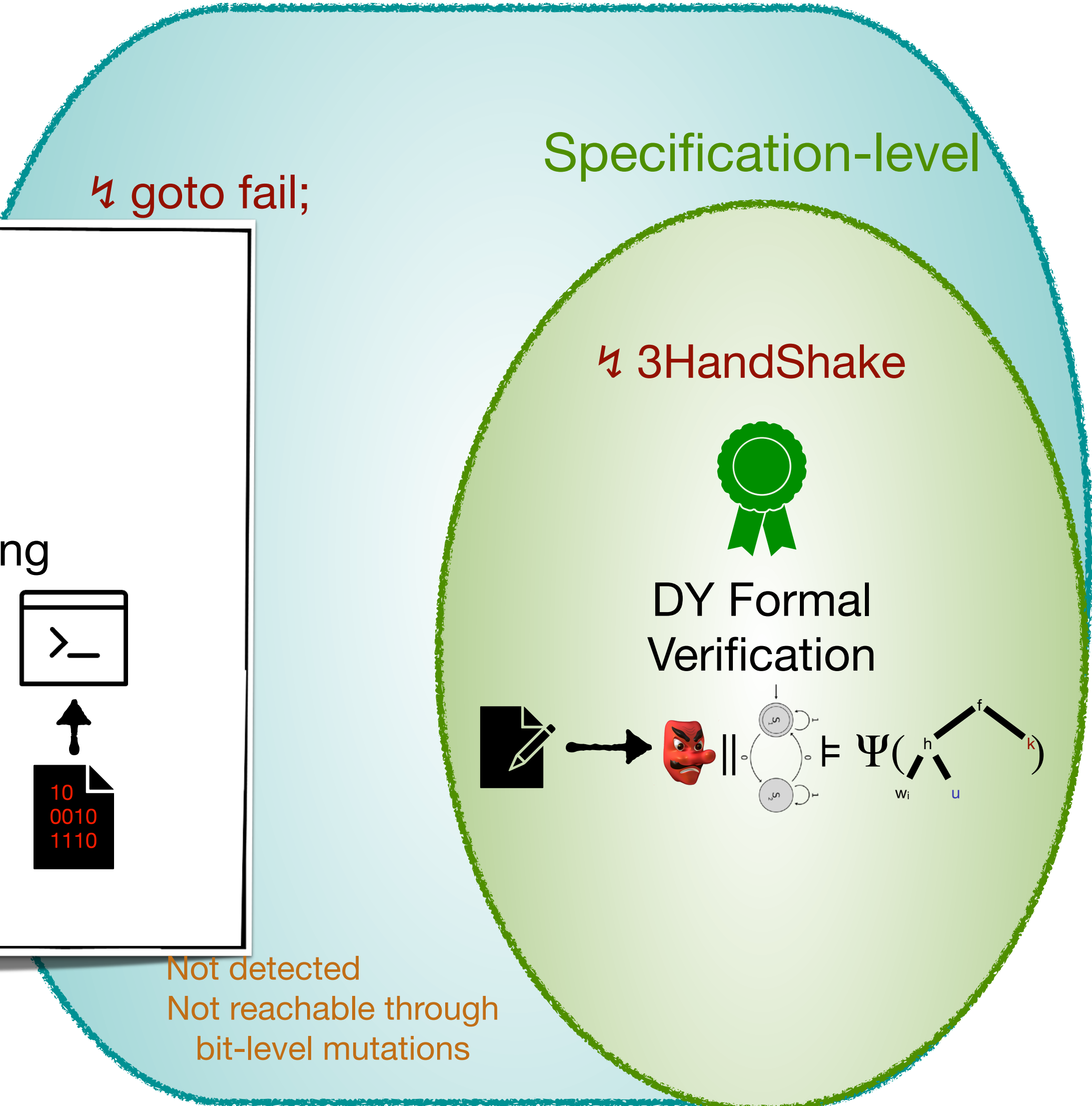


Memory Safety Vulnerabilities

Protocol Vulnerabilities



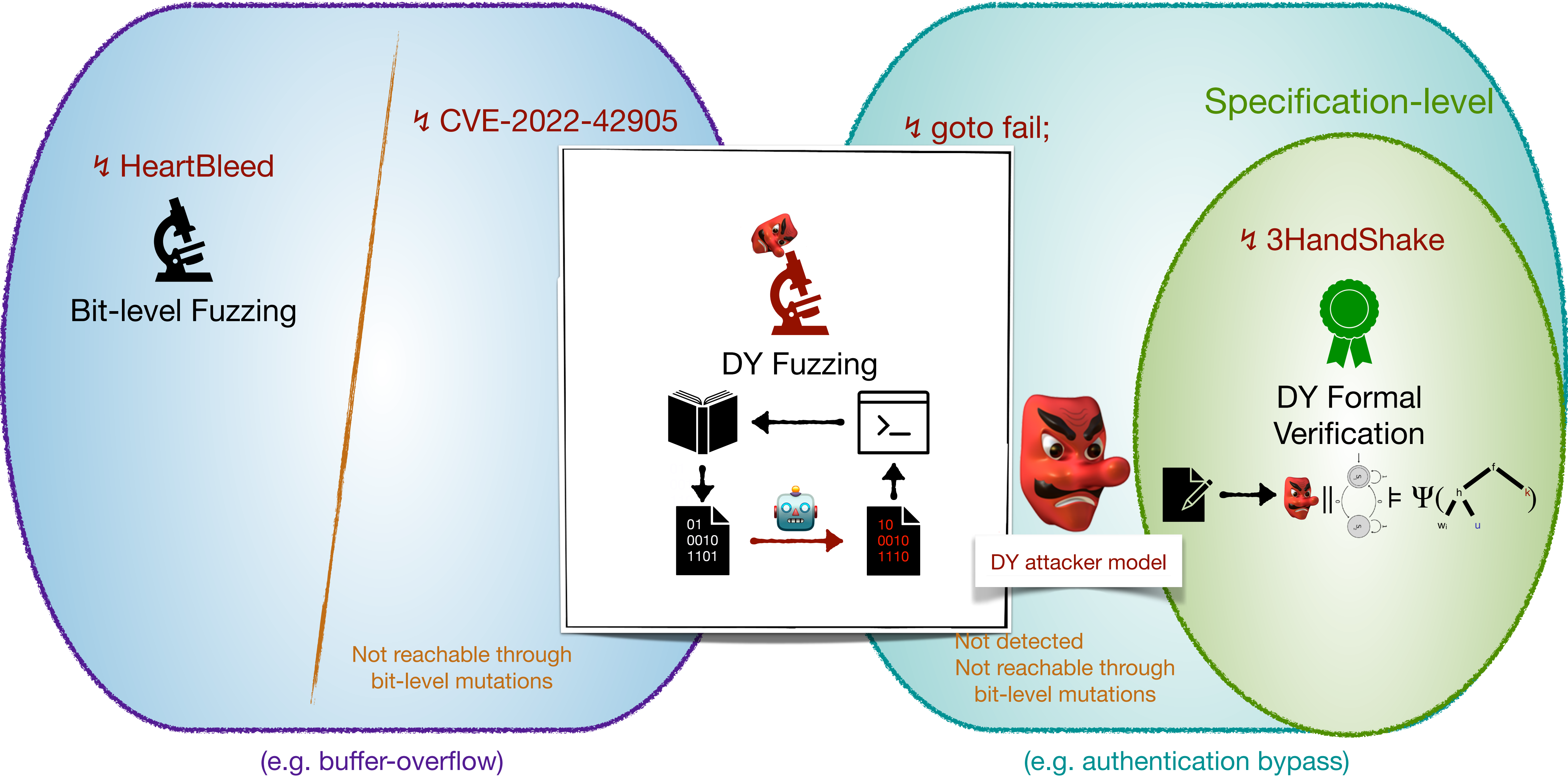
(e.g. buffer-overflow)



(e.g. authentication bypass)

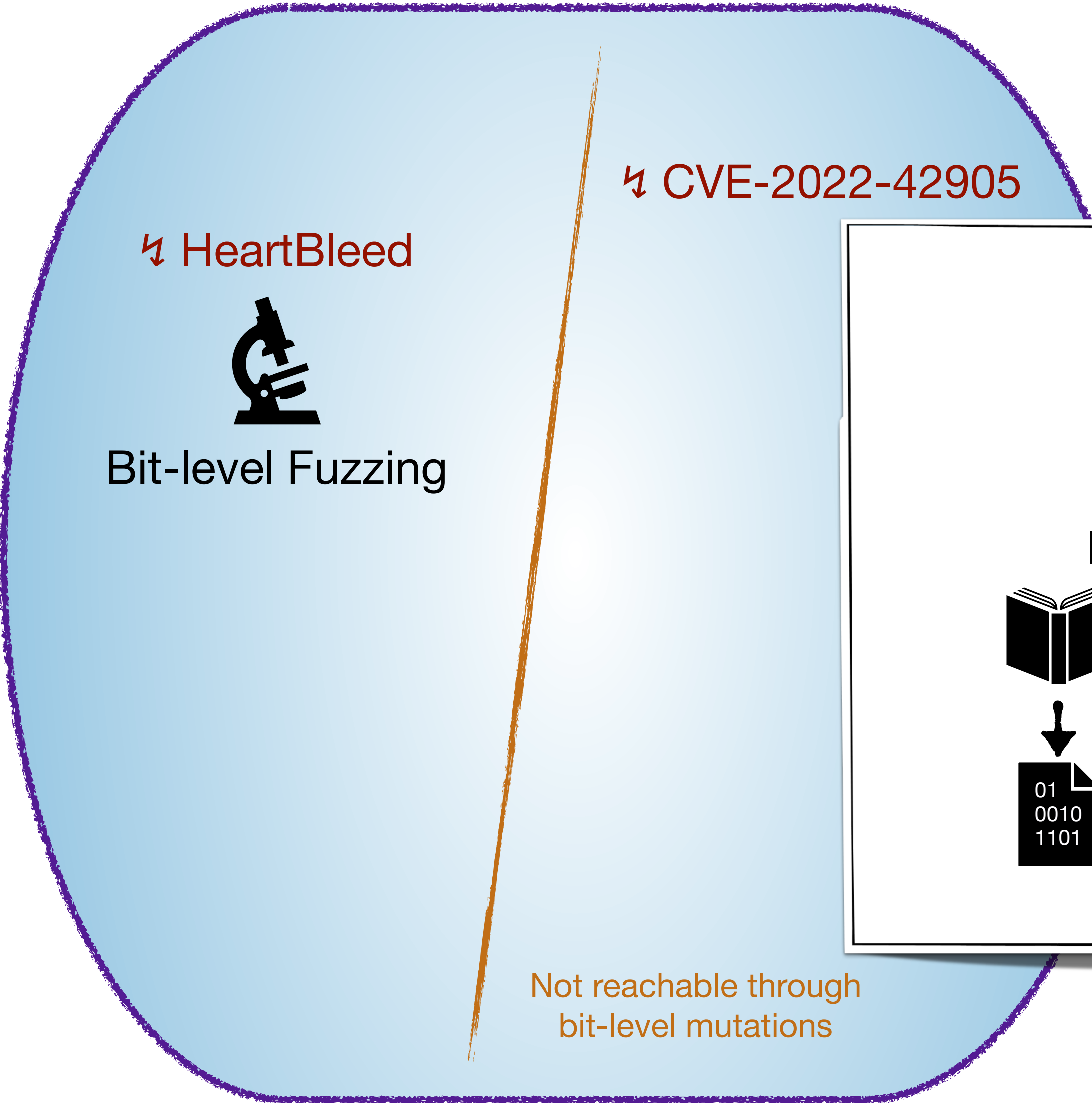
Memory Safety Vulnerabilities

Protocol Vulnerabilities



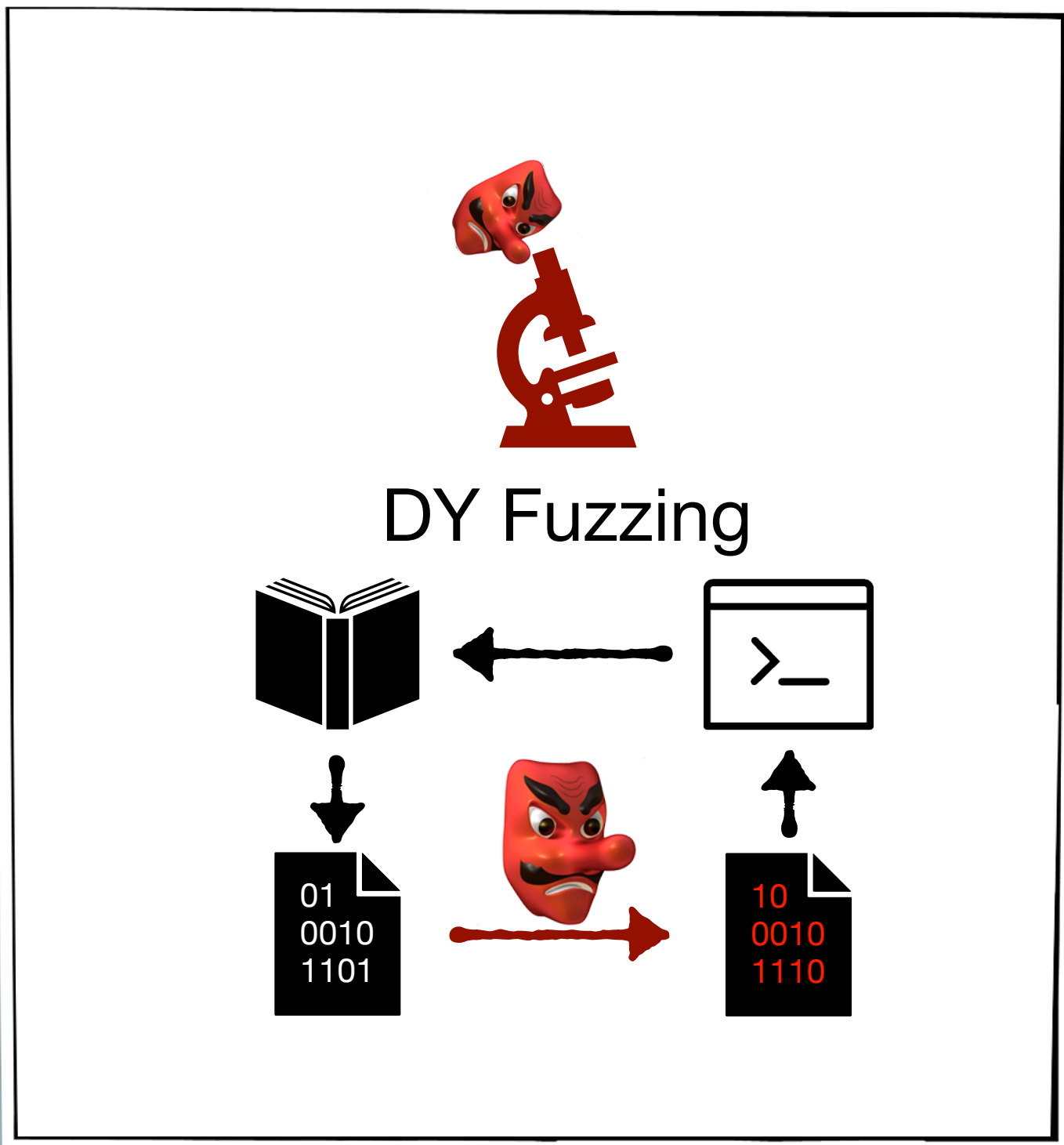
Memory Safety Vulnerabilities

Protocol Vulnerabilities



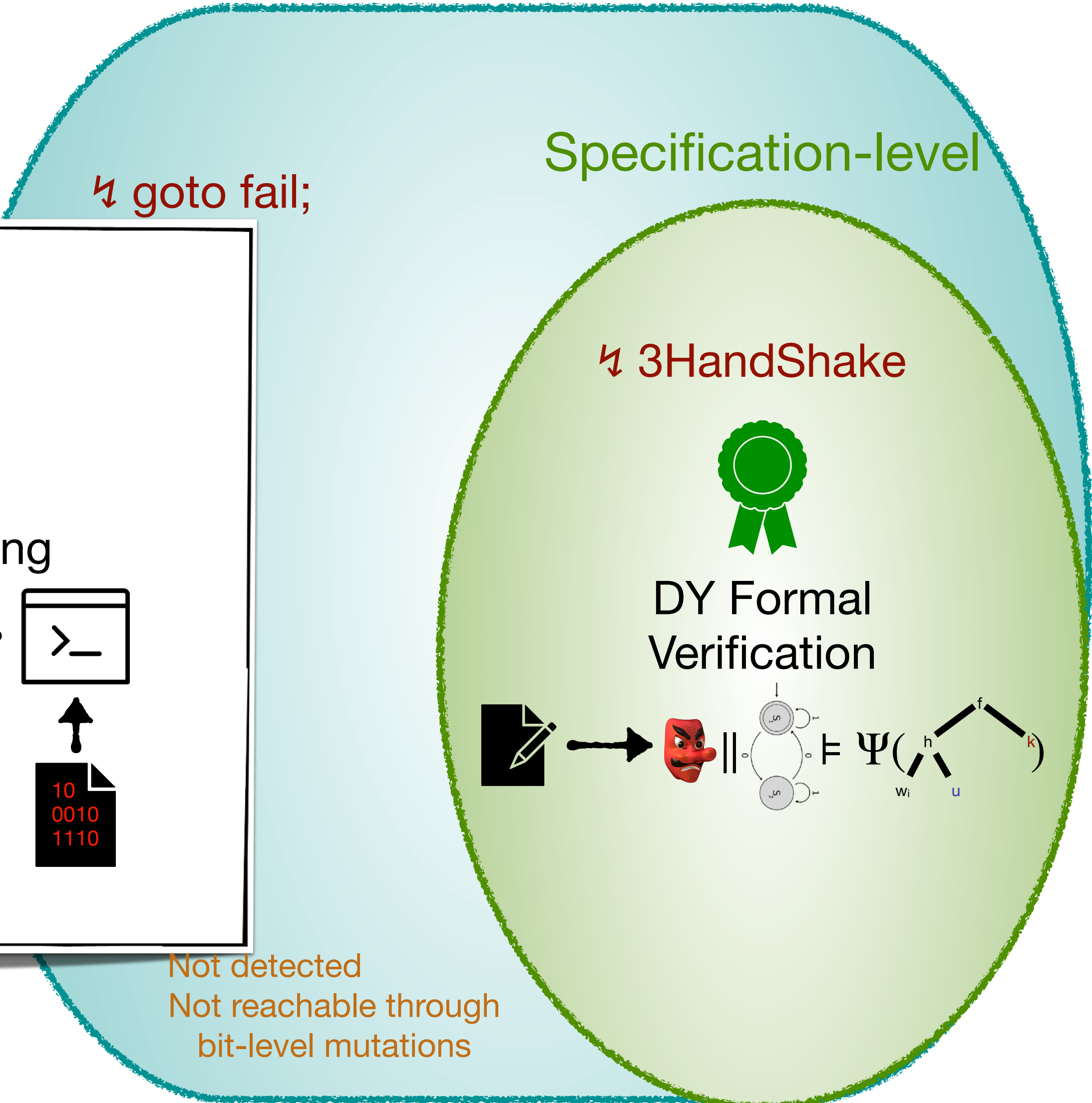
Not reachable through
bit-level mutations

(e.g. buffer-overflow)



Not detected
Not reachable through
bit-level mutations

(e.g. authentication bypass)



Memory Safety Vulnerabilities

Protocol Vulnerabilities

⚡ HeartBleed



Bit-level Fuzzing

⚡ CVE-2022-42905

Not reachable through
bit-level mutations

⚡ goto fail;

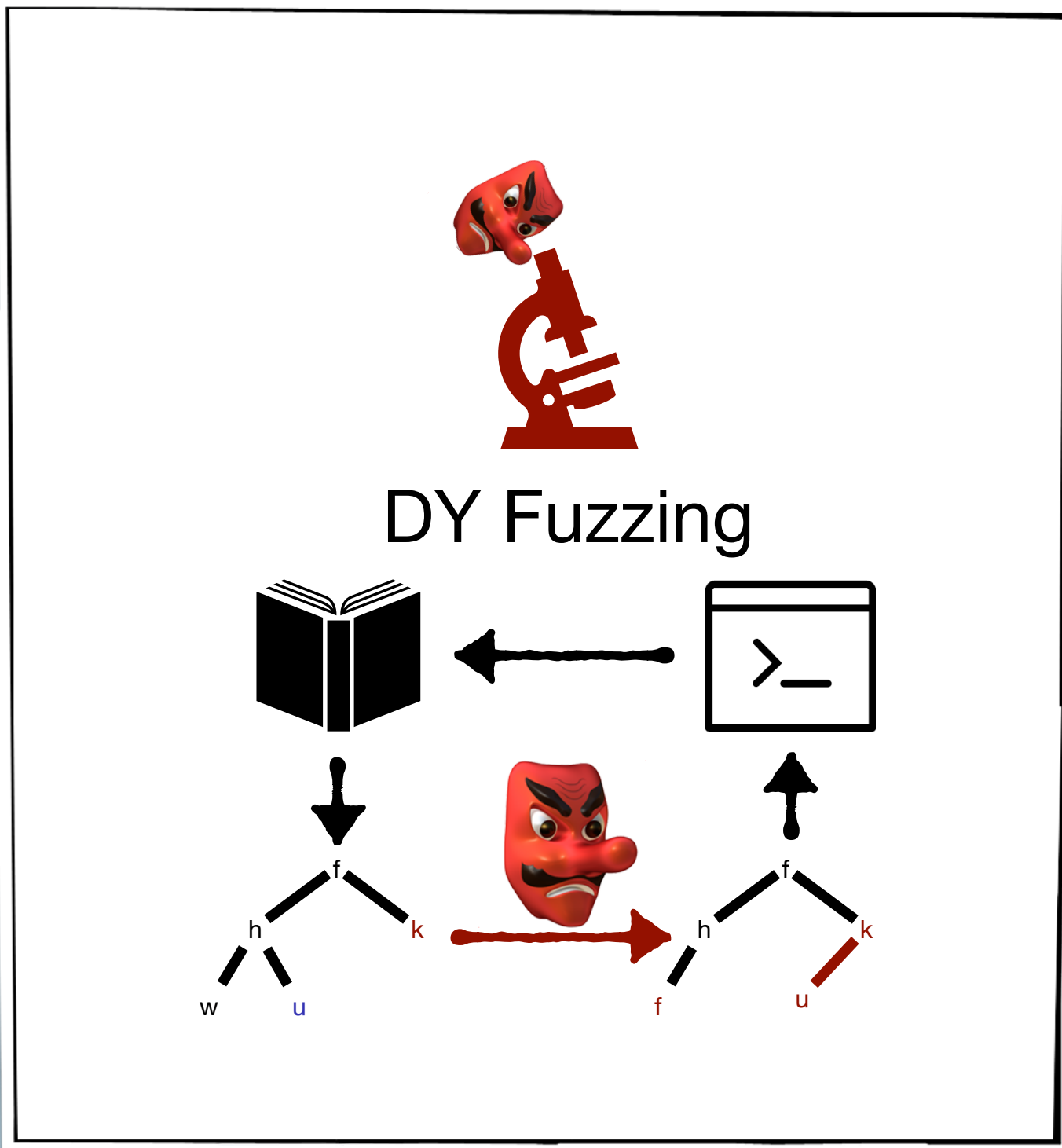
Specification-level

⚡ 3HandShake



DY Formal Verification

Not detected
Not reachable through
bit-level mutations



(e.g. buffer-overflow)

(e.g. authentication bypass)

Memory Safety Vulnerabilities

Protocol Vulnerabilities

⚡ HeartBleed



Bit-level Fuzzing

⚡ CVE-2022-42905

Not reachable through bit-level mutations

⚡ goto fail;

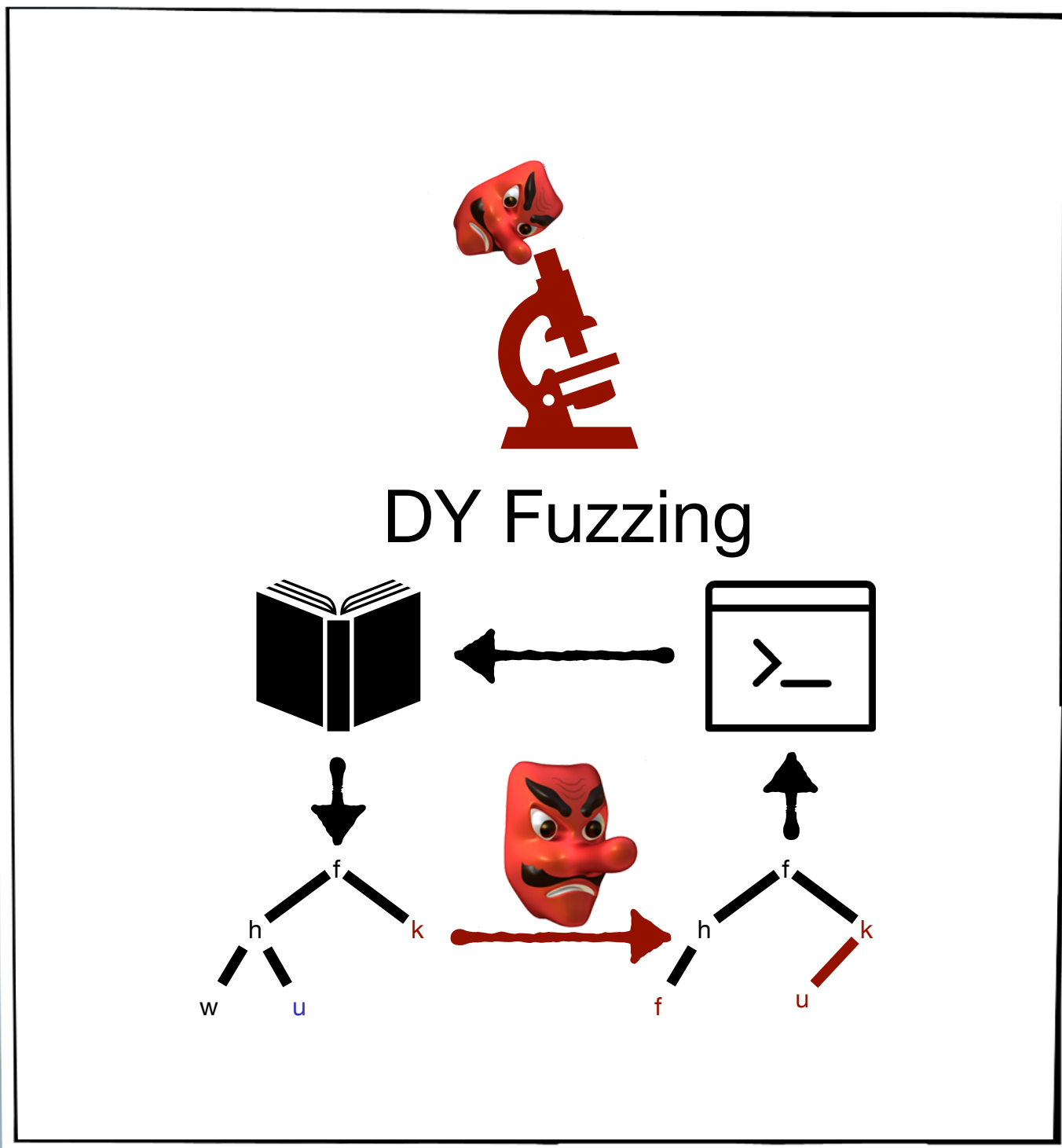
Specification-level

⚡ 3HandShake



DY Formal Verification

Not detected
Not reachable through bit-level mutations

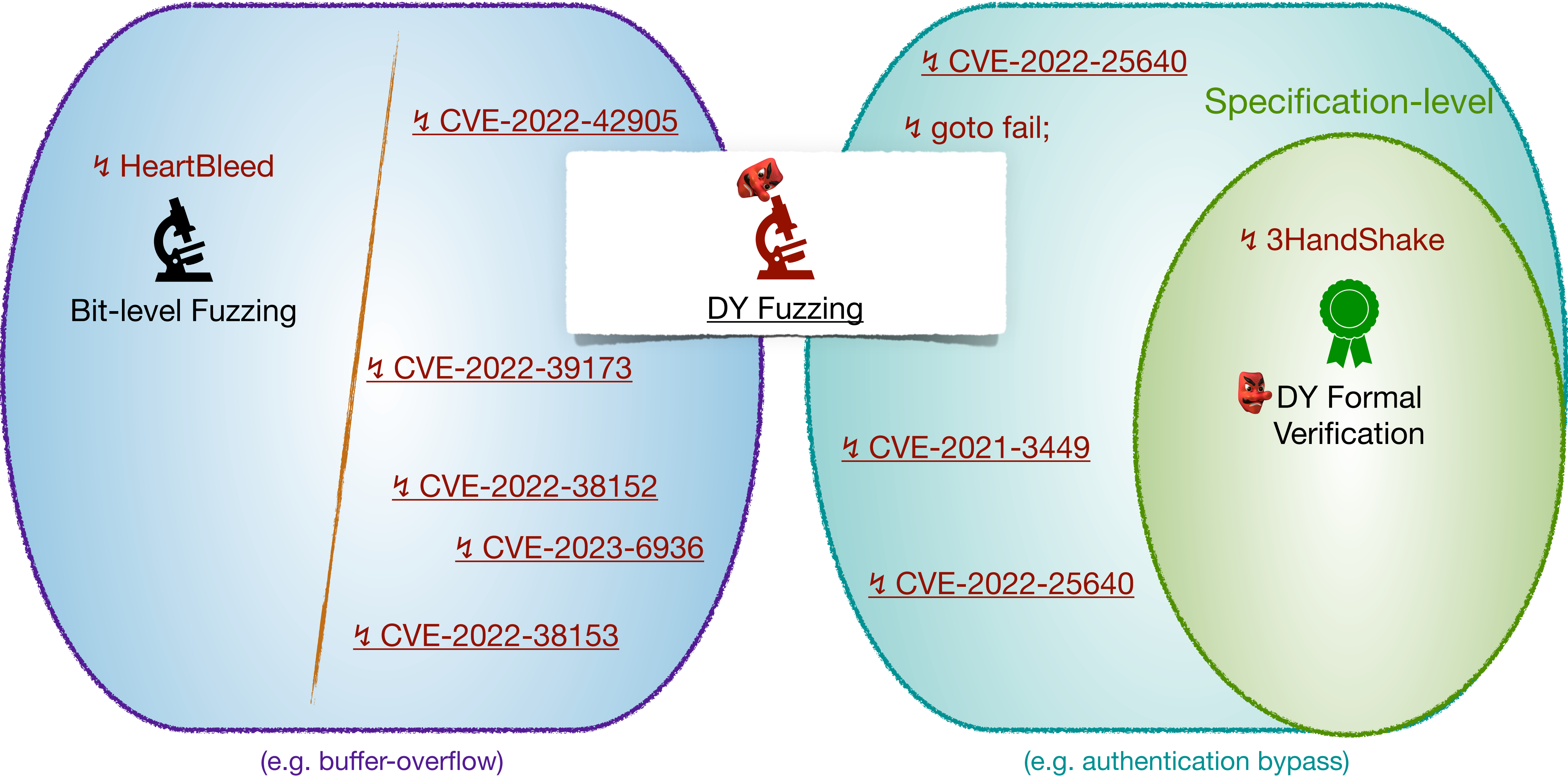


(e.g. buffer-overflow)

(e.g. authentication bypass)

Memory Safety Vulnerabilities

Protocol Vulnerabilities



DY Fuzzing: Formal Dolev-Yao Models Meet Cryptographic 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*



IEEE Security & Privacy, 2024

<https://eprint.iacr.org/2023/57>

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 set of function symbols
Example: $\text{dec}(\circ, \circ)$, $\text{enc}(\circ, \circ)$, $\text{sign}(\circ, \circ)$

DY Fuzzing: Big Picture

DY Fuzzer = DY attacker 🧑‍🦹 in a fuzzing loop

- We build on « messages as formal terms » and assume a set of function symbols

Example: $\text{dec}(\circ, \circ)$, $\text{enc}(\circ, \circ)$, $\text{sign}(\circ, \circ)$

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

$\text{tr} := \text{out}(r, w).\text{tr}$: r is a role (client/server) and w is a variable (attacker knows)
| $\text{in}(r, R).\text{tr}$: 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 set of function symbols

Example: $\text{dec}(\circ, \circ)$, $\text{enc}(\circ, \circ)$, $\text{sign}(\circ, \circ)$

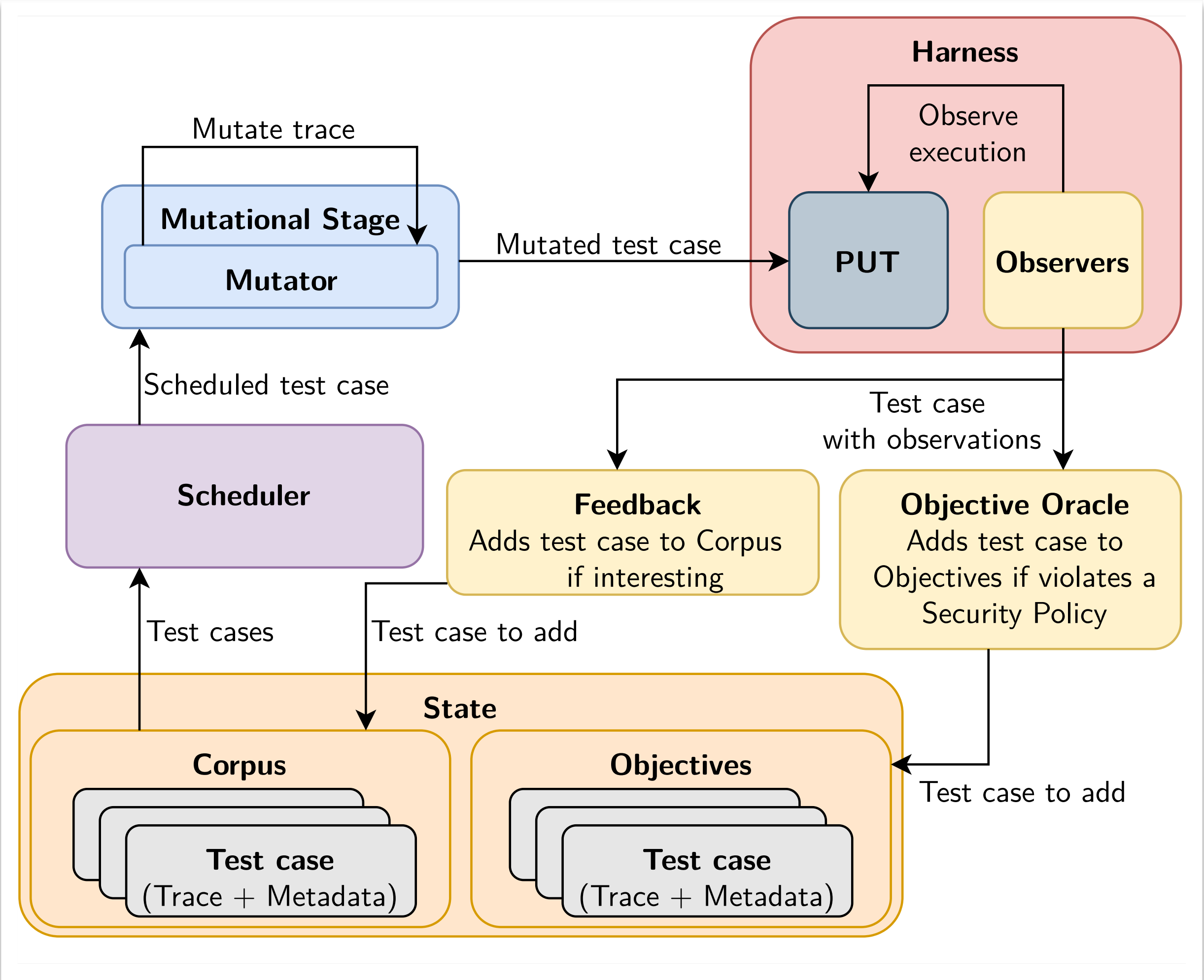
- Test cases = symbolic traces expressing DY attacker 🧑‍🦹's actions

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

Example:

$\text{tr} = \mathbf{out}(\text{client}, w_1).$
 $\mathbf{in}(\text{serv}, w_1).$
 $\mathbf{out}(\text{serv}, w_2).$
 $\mathbf{in}(\text{client}, \text{sign}(\text{dec}(w_2, k_{\text{Bob}}), k_{\text{Att}}))$

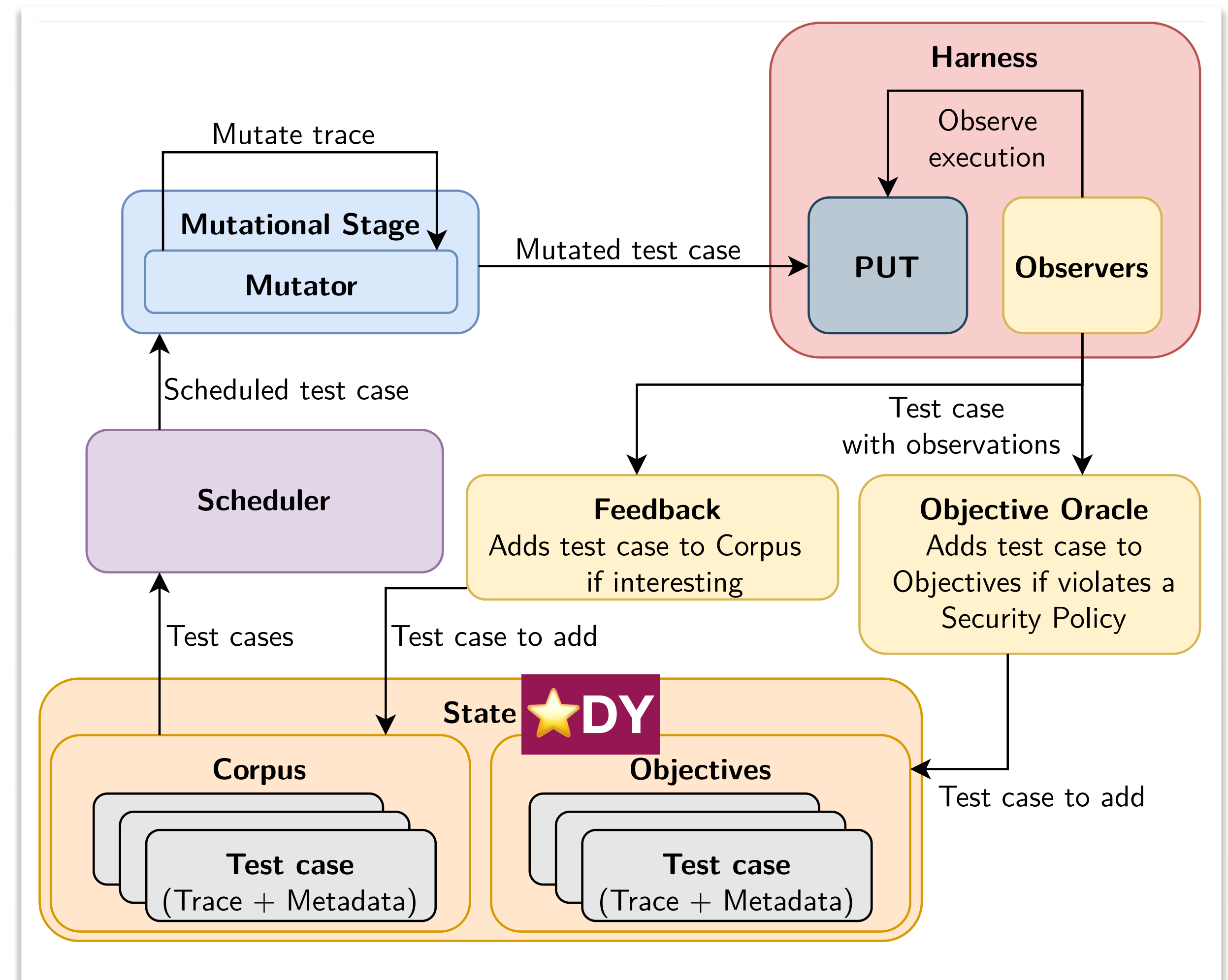
DY Fuzzer Components



LibAFL components (we build on)

DY Fuzzer Components

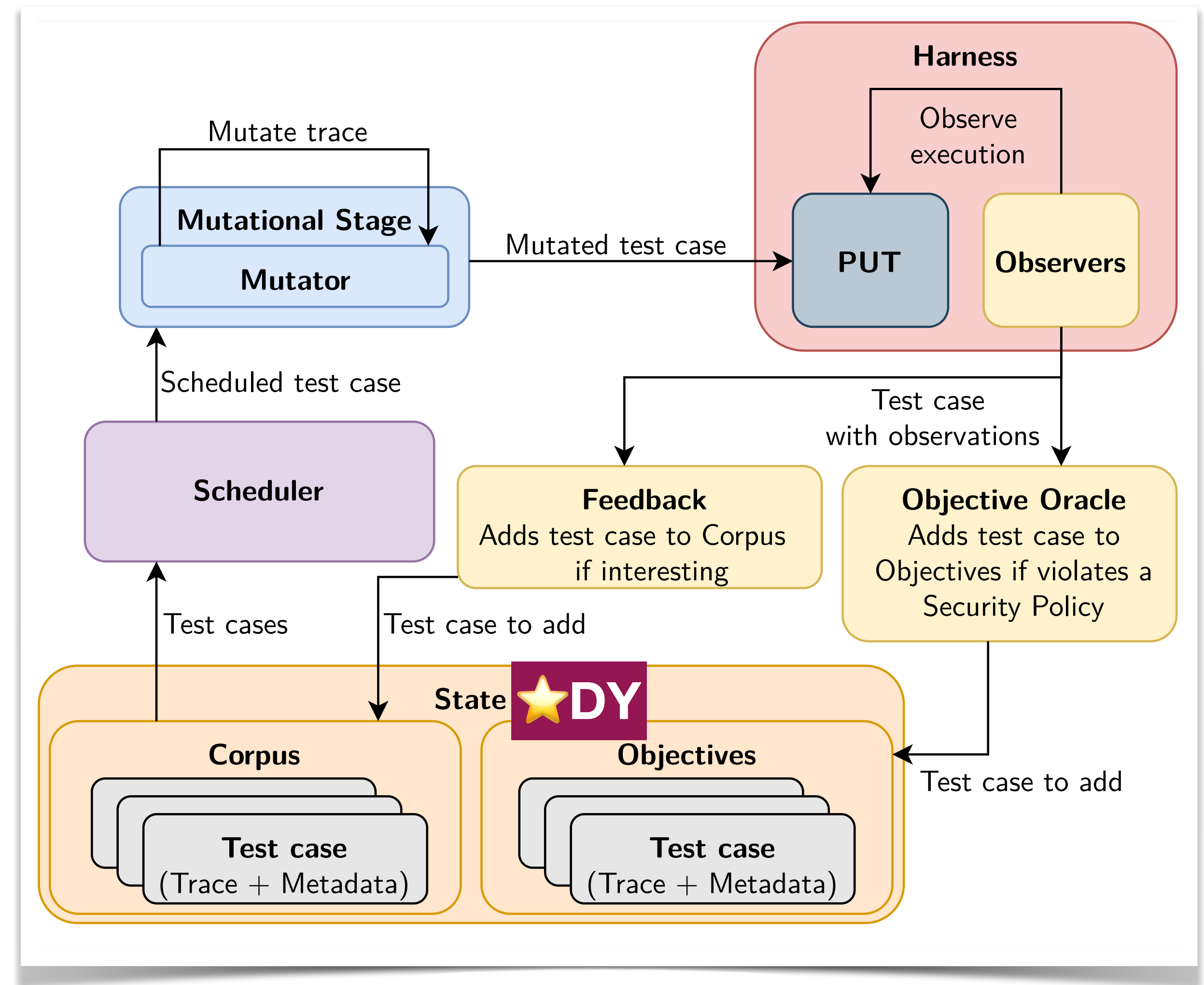
- **State**★: test-cases = DY traces, seeds corpus = happy flows



LibAFL components (we build on)

DY Fuzzer Components

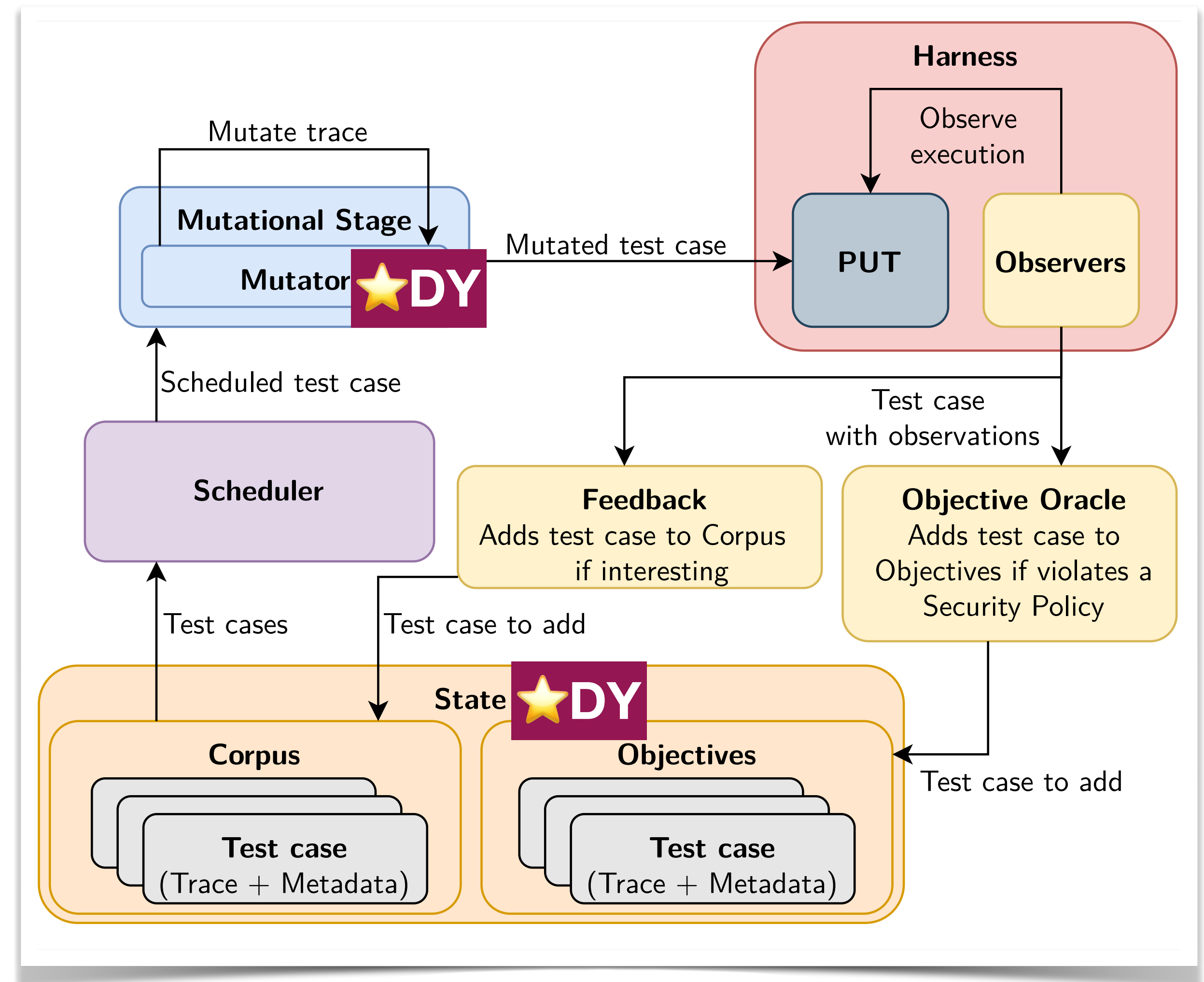
- **State**★: test-cases = DY traces, seeds corpus = happy flows
- **Scheduler**: FIFO



LibAFL components (we build on)

DY Fuzzer Components

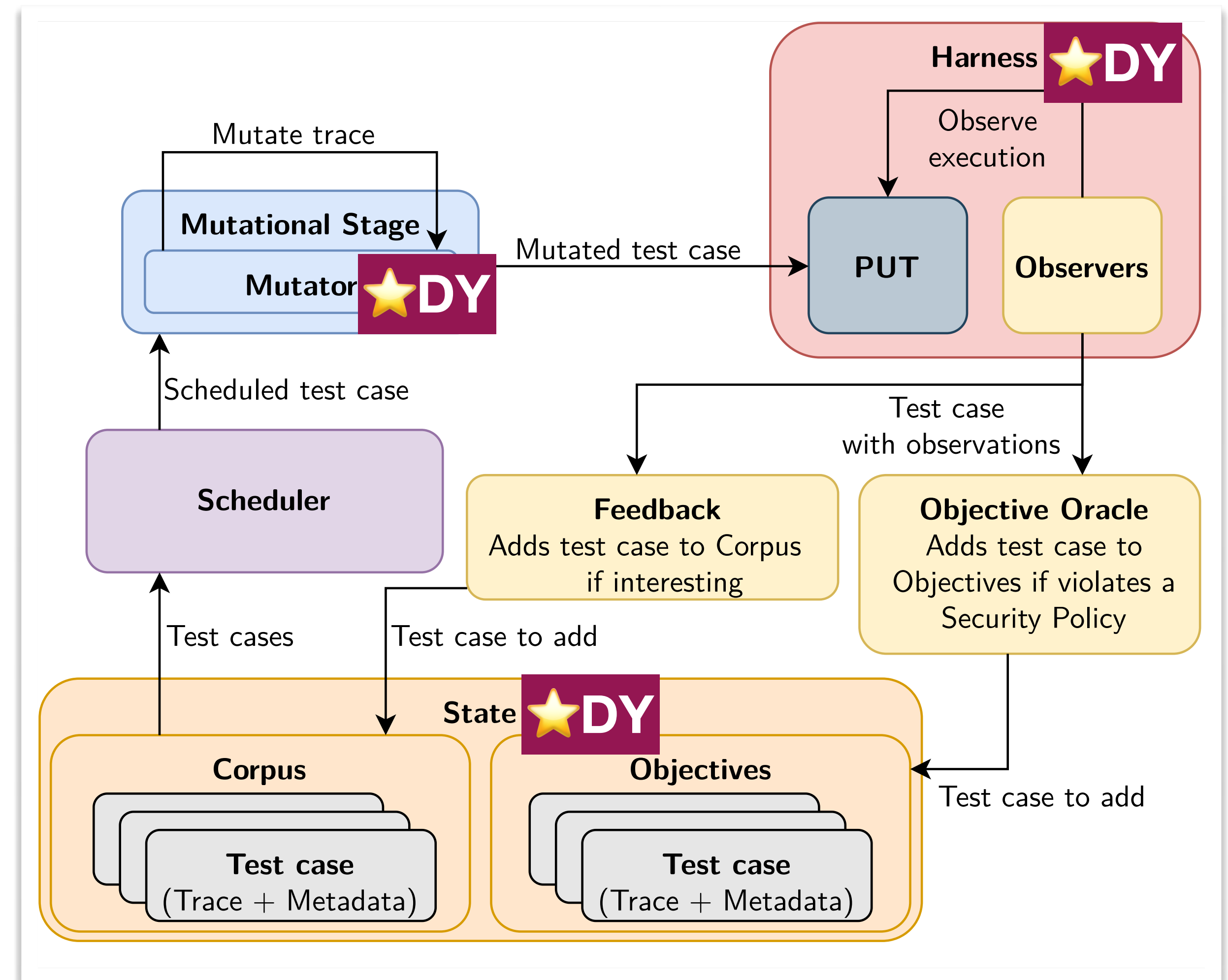
- **State**★: test-cases = DY traces, seeds corpus = happy flows
- **Scheduler**: FIFO
- **Mutator**★: custom trace mutations



LibAFL components (we build on)

DY Fuzzer Components

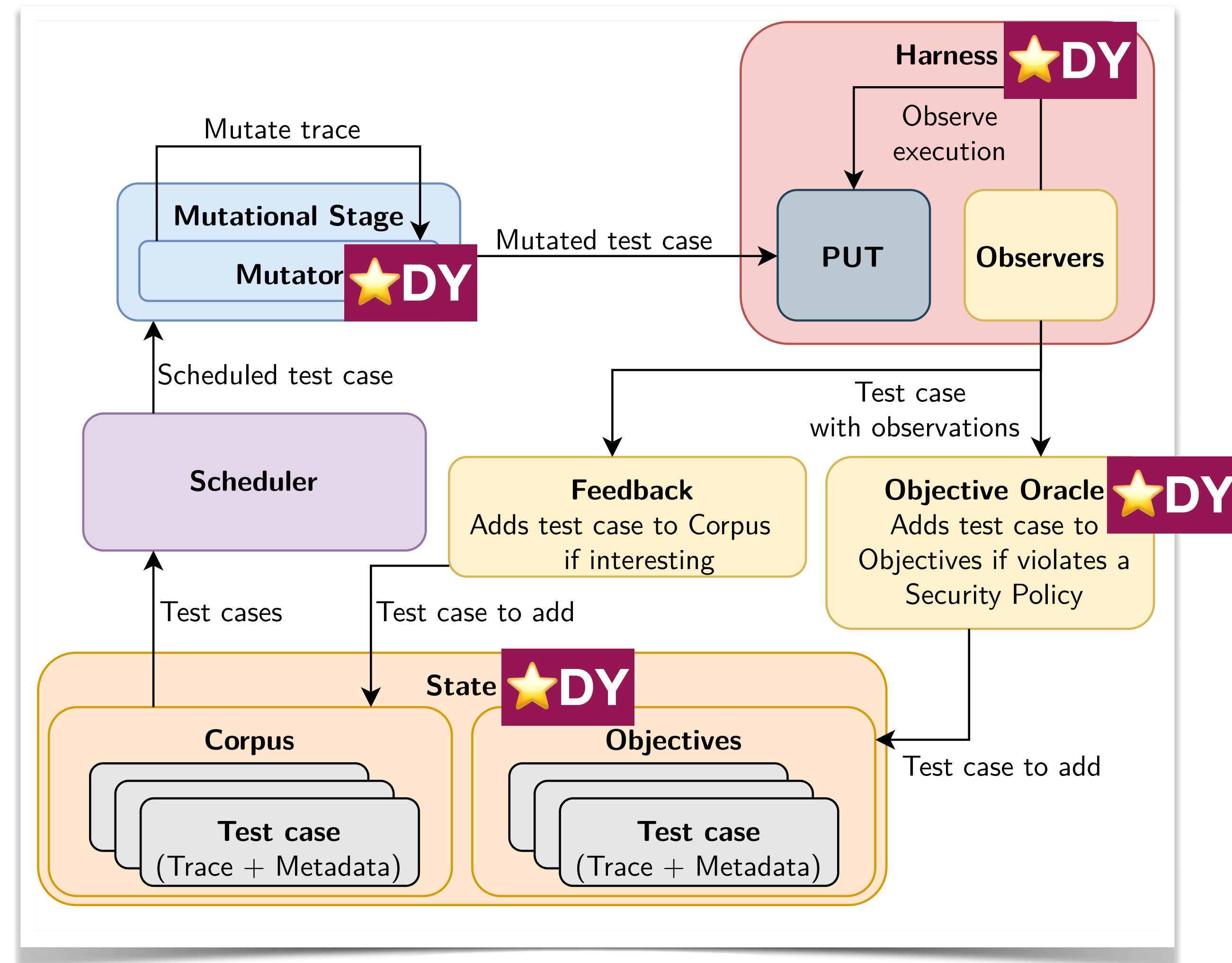
- **State**★: test-cases = DY traces, seeds corpus = happy flows
- **Scheduler**: FIFO
- **Mutator**★: custom trace mutations
- **Harness**★: Mapper + Executor + Claims



LibAFL components (we build on)

DY Fuzzer Components

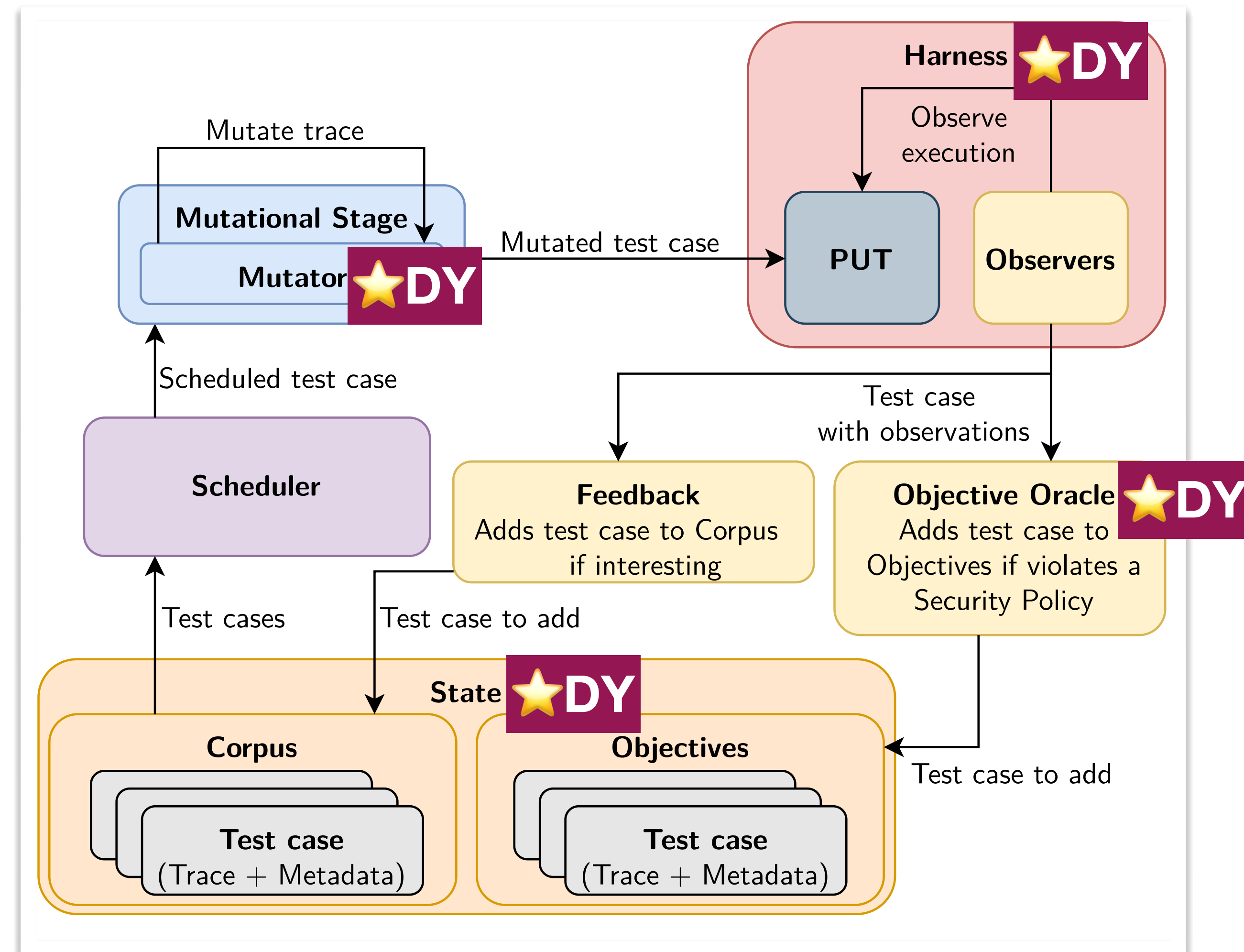
- **State**★: test-cases = DY traces, seeds corpus = happy flows
- **Scheduler**: FIFO
- **Mutator**★: custom trace mutations
- **Harness**★: Mapper + Executor + Claims
- **Obj. Oracle**★: DY security properties★ (e.g. agreement) + ASAN (memory vulns.)



LibAFL components (we build on)

DY Fuzzer Components

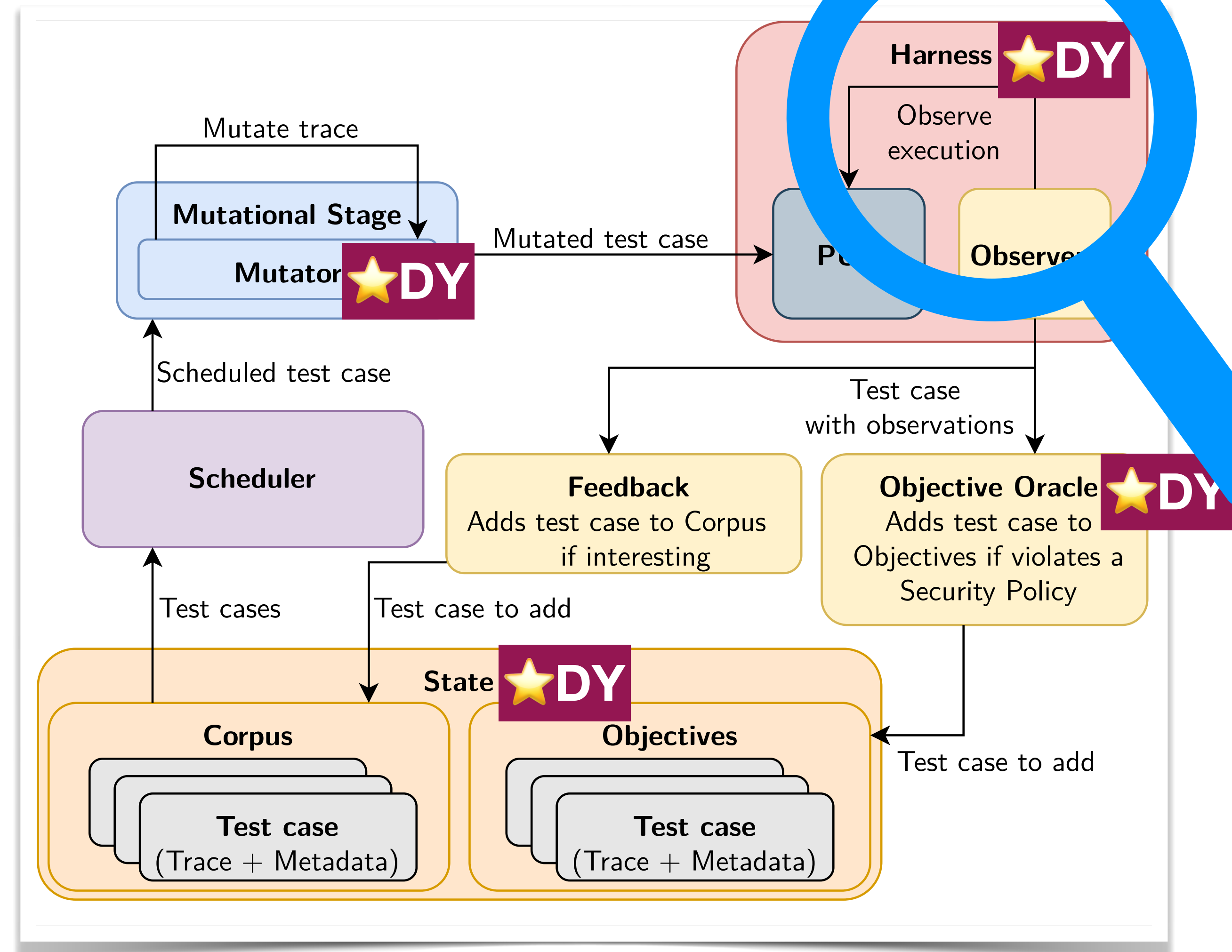
- **State**★: test-cases = DY traces, seeds corpus = happy flows
- **Scheduler**: FIFO
- **Mutator**★: custom trace mutations
- **Harness**★: Mapper + Executor + Claims
- **Obj. Oracle**★: DY security properties★ (e.g. agreement) + ASAN (memory vulns.)
- **Feedback**: PUT code-coverage



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)

Harness 1/2: Mapper

Harness 1/2: Mapper

- We assumed a set of function symbols. Example: $\text{dec}(\circ, \circ)$, $\text{enc}(\circ, \circ)$, $\text{sign}(\circ, \circ)$

Harness 1/2: Mapper

- We assumed a **set of function symbols**. Example: $\text{dec}(\circ, \circ)$, $\text{enc}(\circ, \circ)$, $\text{sign}(\circ, \circ)$
- To each function symbol f , we assume an **interpretation**
$$\llbracket f \rrbracket : [u8]^n \rightarrow [u8] \quad \text{where } [u8] \text{ are bitstrings}$$

Example: $\llbracket \text{sign} \rrbracket(m, \text{key}) := \text{ECDSA}(m, \text{key})$

Harness 1/2: Mapper

- We assumed a **set of function symbols**. Example: $\text{dec}(\circ, \circ)$, $\text{enc}(\circ, \circ)$, $\text{sign}(\circ, \circ)$
- To each function symbol f , we assume an **interpretation**
$$\llbracket f \rrbracket : [u8]^n \rightarrow [u8] \quad \text{where } [u8] \text{ are bitstrings}$$

Example: $\llbracket \text{sign} \rrbracket(m, \text{key}) := \text{ECDSA}(m, \text{key})$

- Constants like k_{Att} are statically generated $\llbracket k_{Att} \rrbracket := \text{genKey}()$

Harness 1/2: Mapper

- We assumed a **set of function symbols**. Example: $\text{dec}(\circ, \circ)$, $\text{enc}(\circ, \circ)$, $\text{sign}(\circ, \circ)$
- To each function symbol f , we assume an **interpretation**
$$\llbracket f \rrbracket : [u8]^n \rightarrow [u8] \quad \text{where } [u8] \text{ are bitstrings}$$

Example: $\llbracket \text{sign} \rrbracket(m, \text{key}) := \text{ECDSA}(m, \text{key})$

- Constants like k_{Att} are statically generated $\llbracket k_{Att} \rrbracket := \text{genKey}()$
- 👉 **Mapper** can **interpret any term** by recursively applying interpretations

$$\llbracket \cdot \rrbracket : \mathcal{T}\text{erms} \rightarrow [u8]$$

Example: $\llbracket \text{sign}(\text{dec}(w, k_{Bob}), k_{Att}) \rrbracket := \llbracket \text{sign} \rrbracket(\llbracket \text{dec}(w, k_{Bob}) \rrbracket, \llbracket k_{Att} \rrbracket)$

Harness 1/2: Mapper

- We assumed a **set of function symbols**. Example: $\text{dec}(\circ, \circ)$, $\text{enc}(\circ, \circ)$, $\text{sign}(\circ, \circ)$
- To each function symbol f , we assume an **interpretation**
$$\llbracket f \rrbracket : [u8]^n \rightarrow [u8] \quad \text{where } [u8] \text{ are bitstrings}$$

Example: $\llbracket \text{sign} \rrbracket(m, \text{key}) := \text{ECDSA}(m, \text{key})$

- Constants like k_{Att} are statically generated $\llbracket k_{Att} \rrbracket := \text{genKey}()$
- 👉 **Mapper** can **interpret any term** by recursively applying interpretations

$$\llbracket \cdot \rrbracket : \mathcal{T}\text{erms} \rightarrow [u8]$$

Example: $\llbracket \text{sign}(\text{dec}(w, k_{Bob}), k_{Att}) \rrbracket := \llbracket \text{sign} \rrbracket(\llbracket \text{dec}(w, k_{Bob}) \rrbracket, \llbracket k_{Att} \rrbracket)$

Mapper is **protocol-dependent** but **PUT-independent** and can be built **once-for-all** on top of a reference implementation or *any* PUT

Harness 2/2: **Executor**

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

Harness 2/2: **Executor**

Executor concretizes DY traces (tr) with the PUT (e.g. OpenSSL):

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

Harness 2/2: **Executor**

Executor concretizes DY traces (tr) with the PUT (e.g. OpenSSL):

1. Initialize all agents, **client** and **serv**, and their IO buffers

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

Harness 2/2: Executor

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

Executor concretizes DY traces (tr) with the PUT (e.g. OpenSSL):

1. Initialize all agents, *client* and *serv*, and their IO buffers
2. On output actions: e.g. **out**(*client*, w)
 - a. call **PUT** to read bitstring b_w from output buffer of *client*
 - b. let *client* progress

Harness 2/2: Executor

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

Executor concretizes DY traces (tr) with the PUT (e.g. OpenSSL):

1. Initialize all agents, *client* and *serv*, and their IO buffers
2. On output actions: e.g. **out**(*client*, w)
 - a. call **PUT** to read bitstring b_w from output buffer of *client*
 - b. let *client* progress
3. On input actions: e.g. **in**(*serv*, R)
 - c. invoke **Mapper** to concretise term R into a bitstring $b_R := \llbracket R \rrbracket$
 - d. call **PUT** to write b_R onto input buffer of *serv*
 - e. let *serv* progress

Harness 2/2: Executor

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

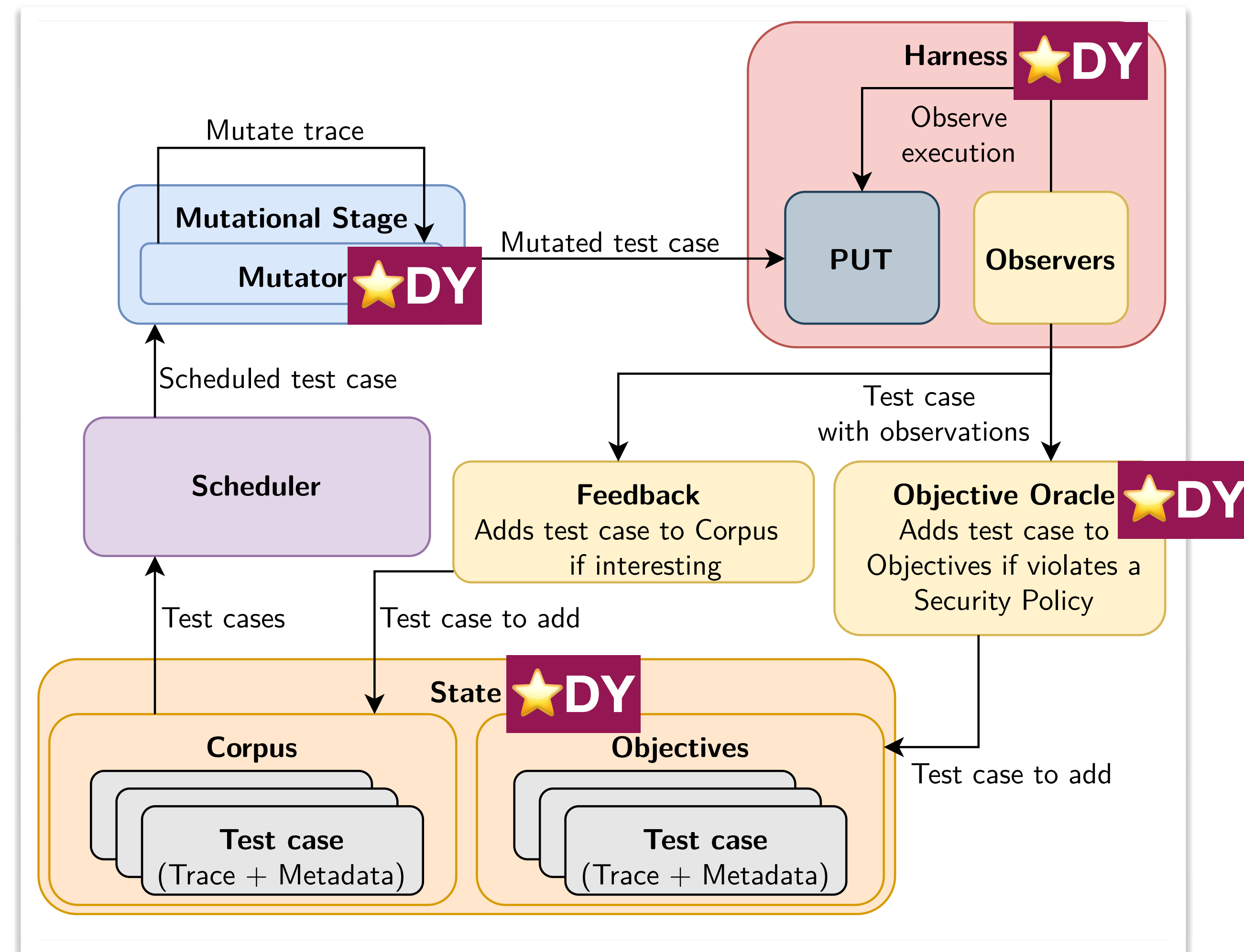
Executor concretizes DY traces (tr) with the PUT (e.g. OpenSSL):

1. Initialize all agents, *client* and *serv*, and their IO buffers
2. On output actions: e.g. **out**(*client*, w)
 - a. call **PUT** to read bitstring b_w from output buffer of *client*
 - b. let *client* progress
3. On input actions: e.g. **in**(*serv*, R)
 - c. invoke **Mapper** to concretise term R into a bitstring $b_R := \llbracket R \rrbracket$
 - d. call **PUT** to write b_R onto input buffer of *serv*
 - e. let *serv* progress

Executor is PUT-dependent but requires lightweight harnessing

DY Fuzzer Components

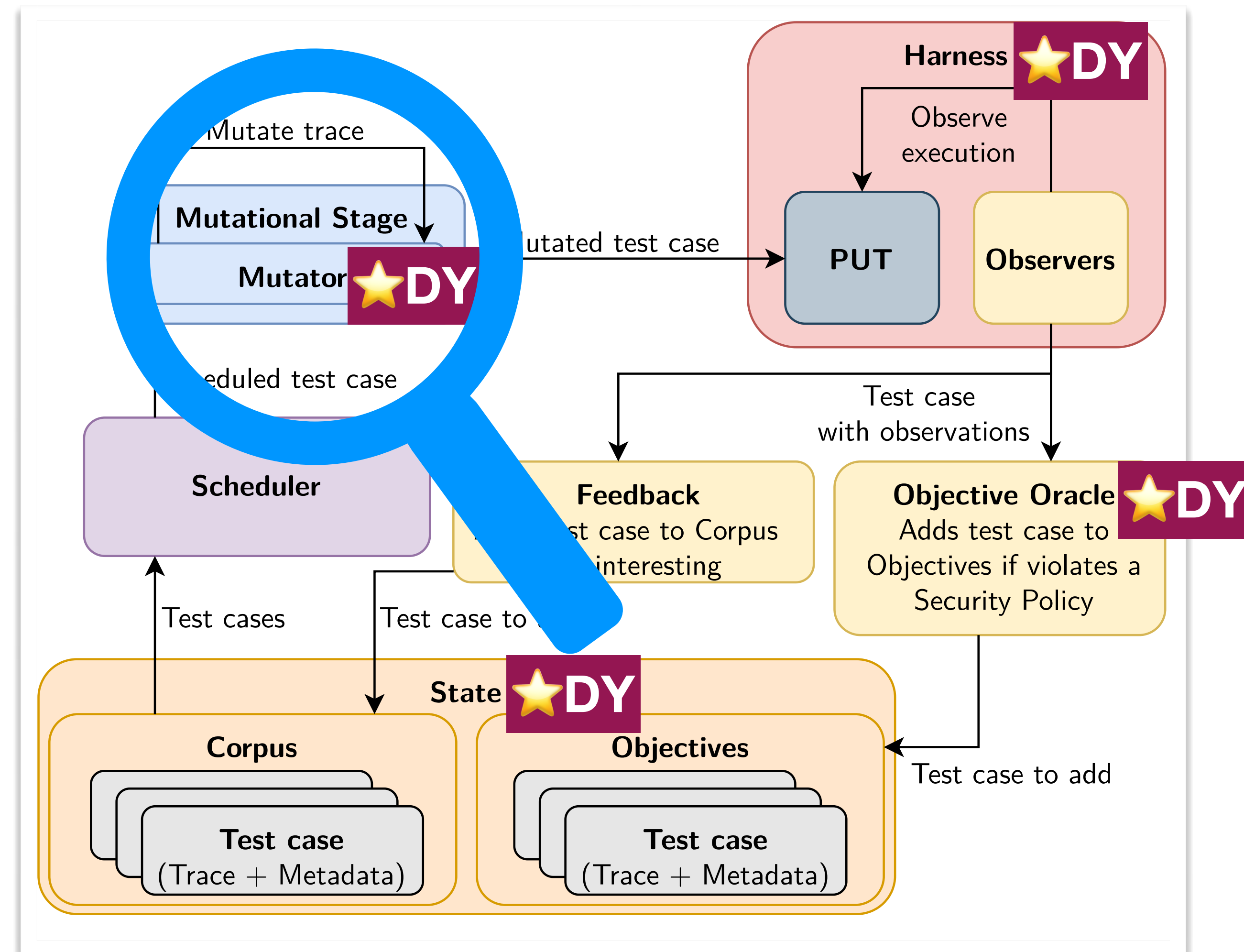
- **State**★: test-cases = DY traces, seeds corpus = happy flows
- **Scheduler**: FIFO
- **Mutator**★: custom trace mutations
- **Harness**★: Mapper + Executor + Claims
- **Obj. Oracle**★: DY security properties★ (e.g. agreement) + ASAN (memory vulns.)
- **Feedback**: PUT code-coverage



LibAFL components (we build on)

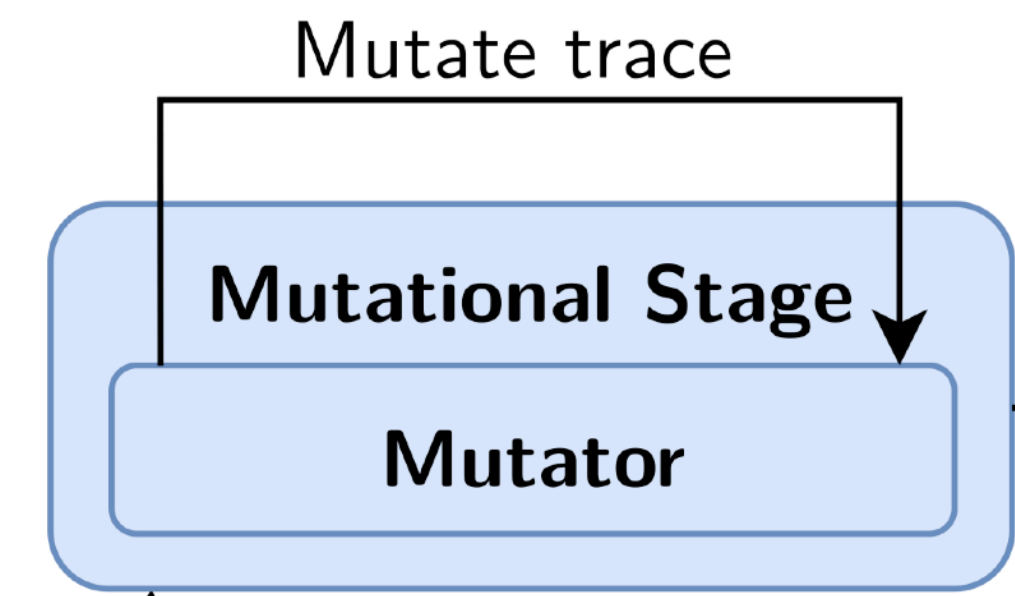
DY Fuzzer Components

- **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

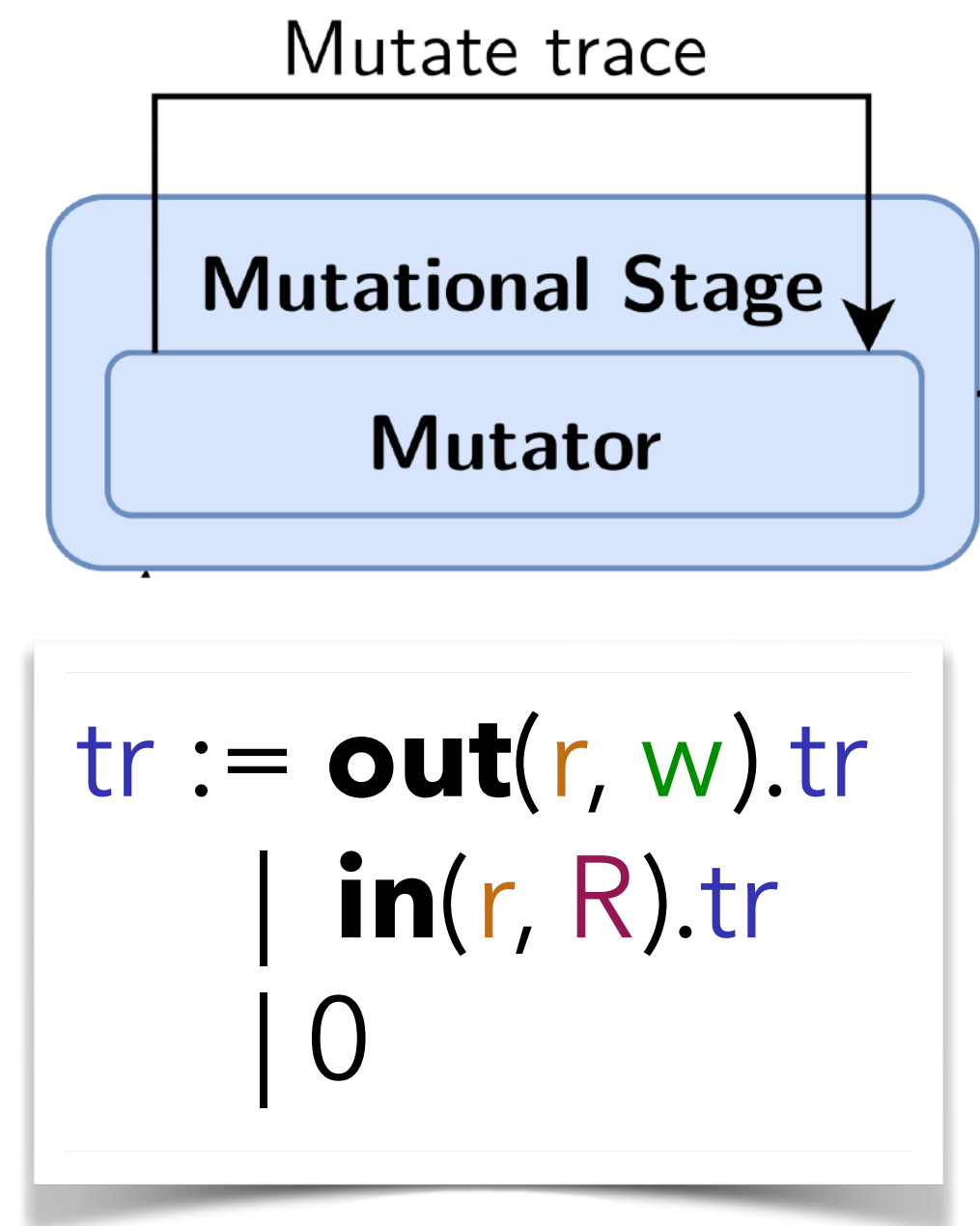


```
tr := out(r, w).tr  
    | in(r, R).tr  
    | 0
```

DY mutations

Action-level Mutations

Term-level Mutations★

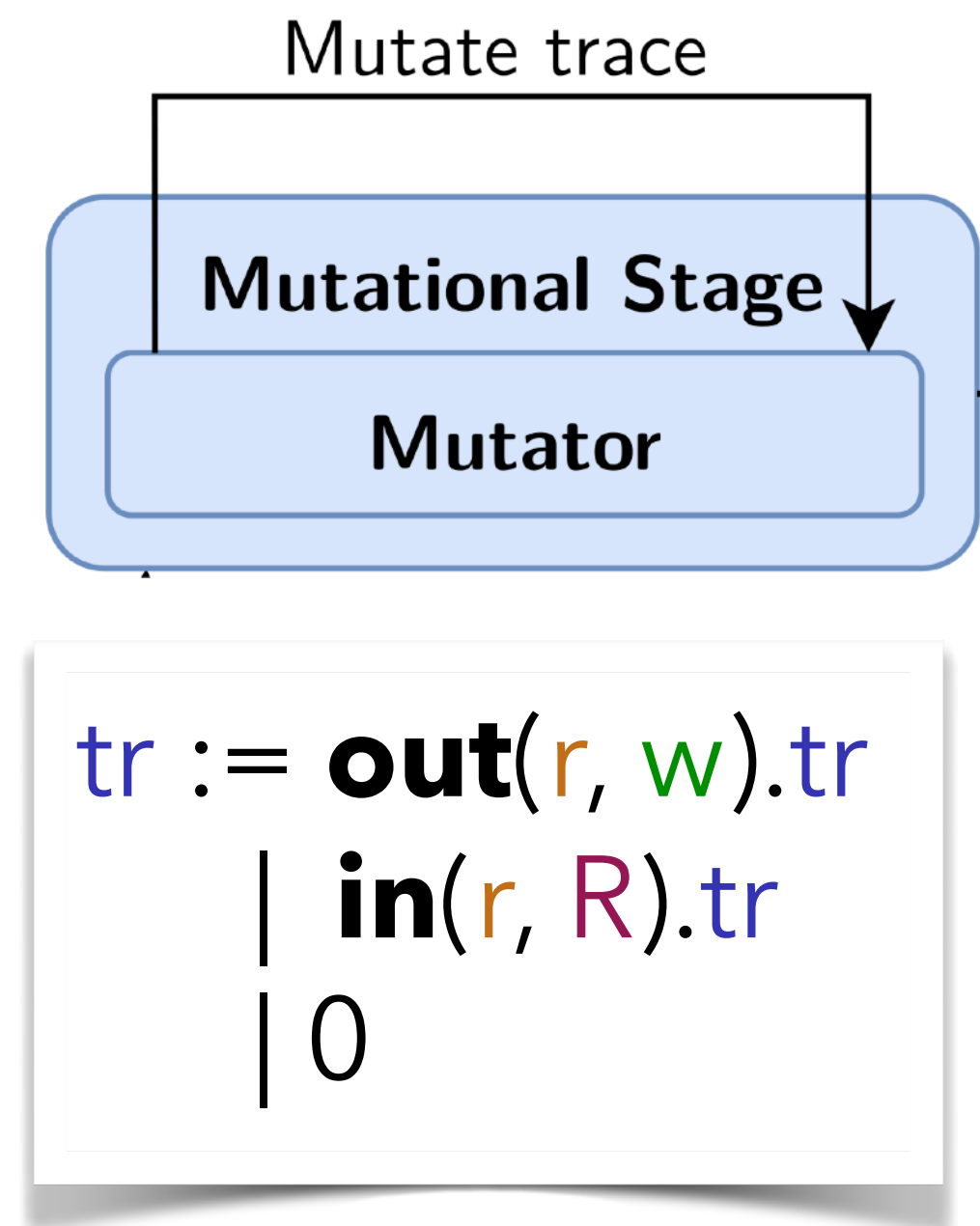


DY mutations

Action-level Mutations

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

Term-level Mutations★



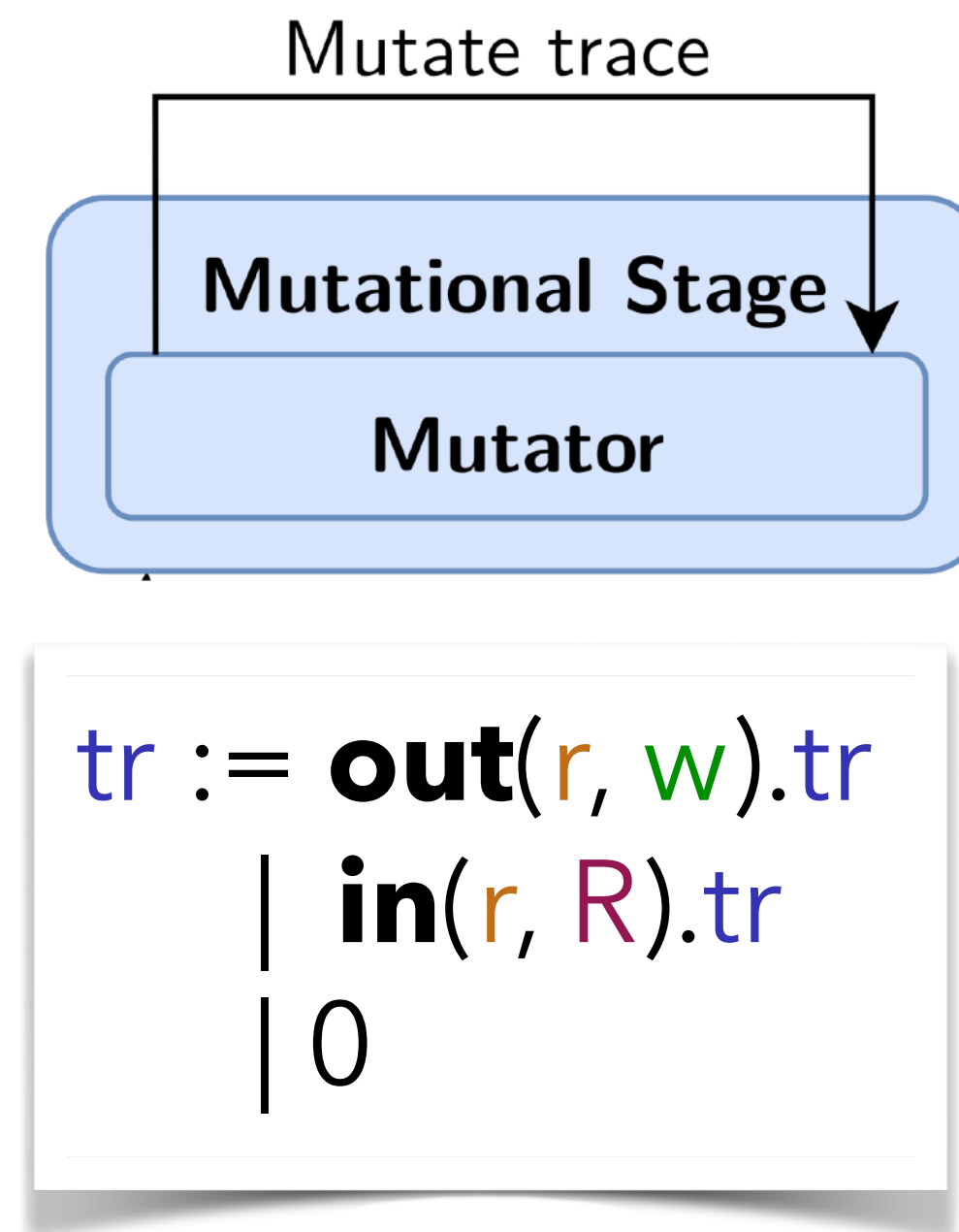
DY mutations

Action-level Mutations

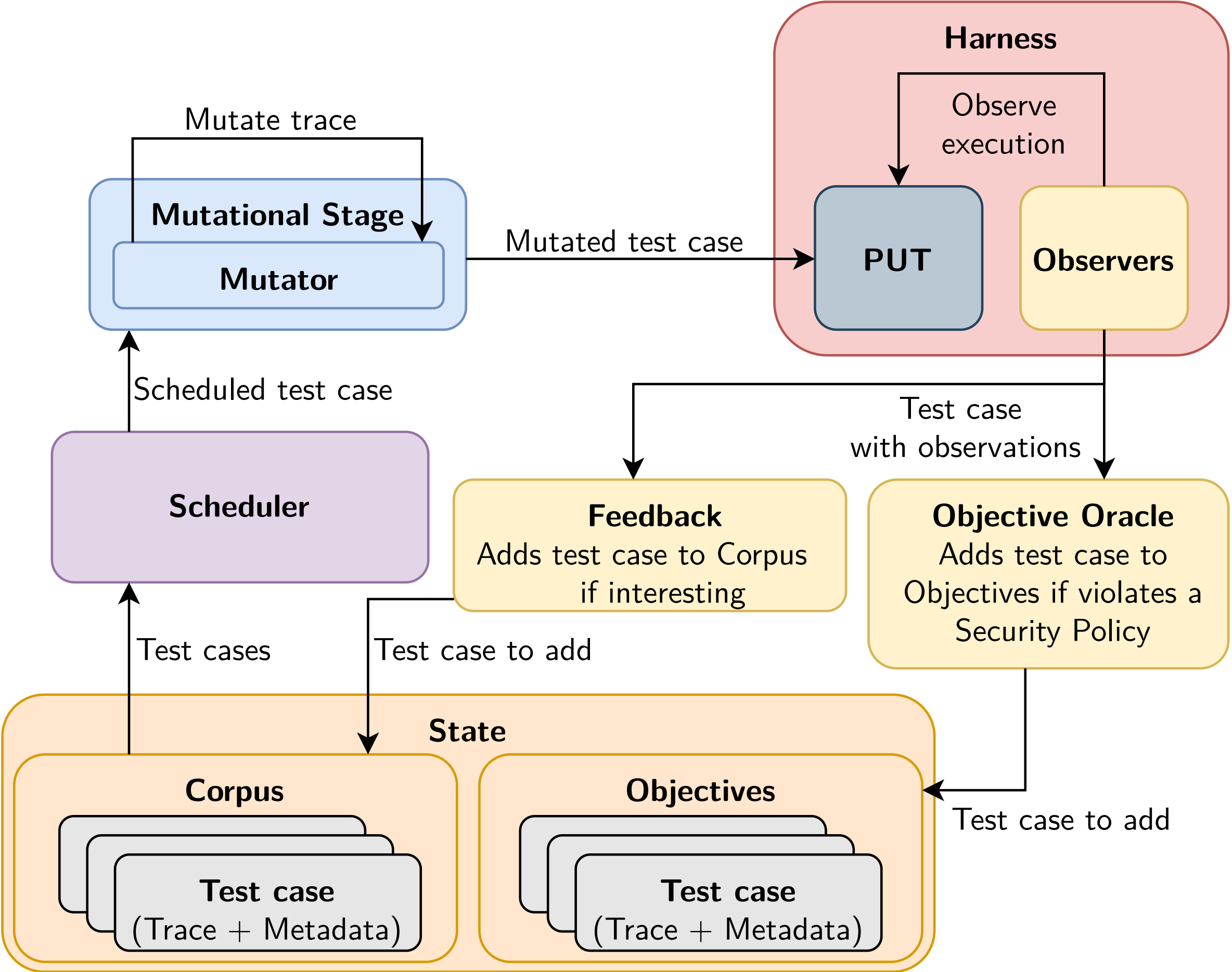
- **Skip**: remove random action (in/out)
- **Repeat**: randomly copy and insert an action

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 $\leftrightarrow$ 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



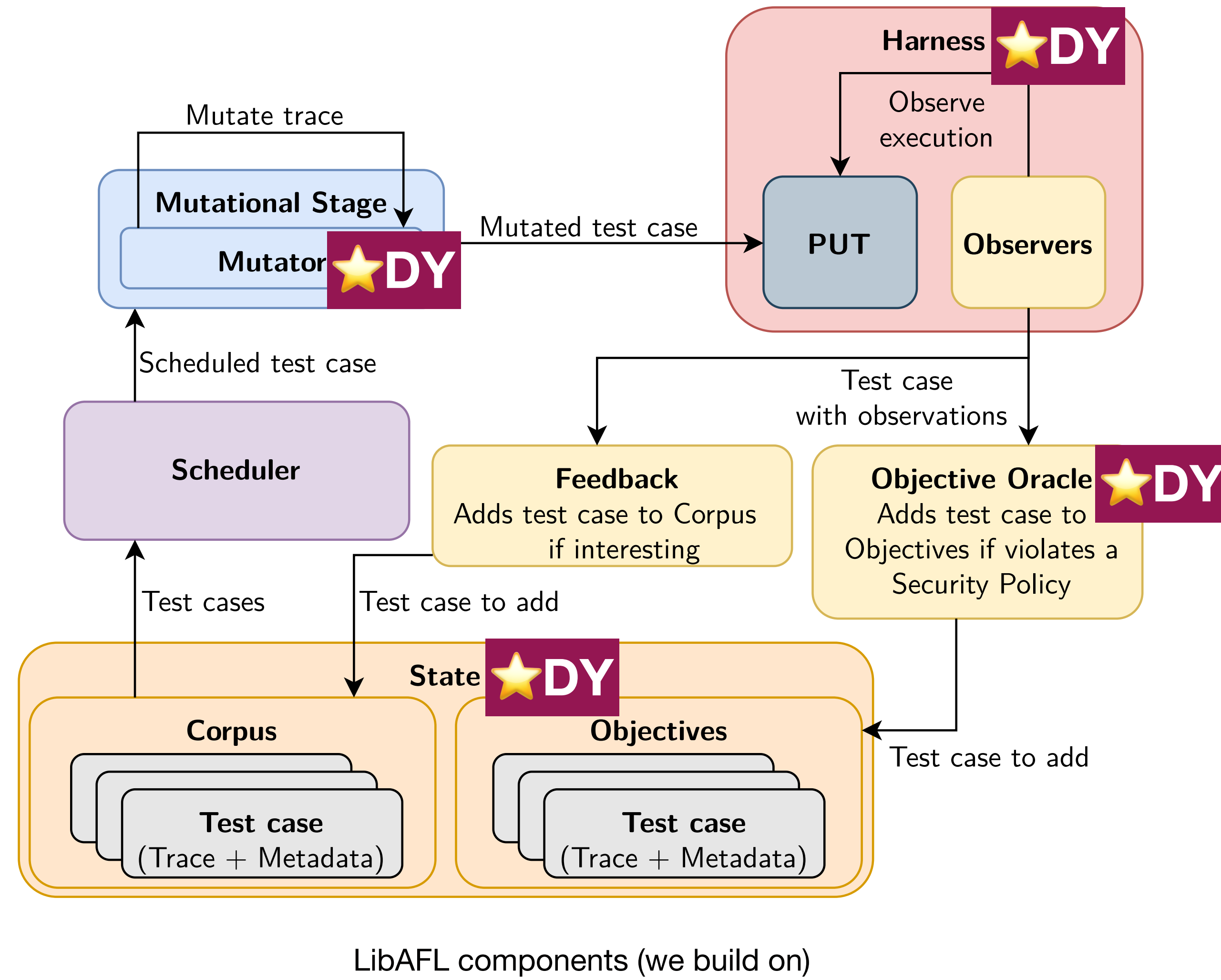
DY Fuzzer components



LibAFL components (we build on)

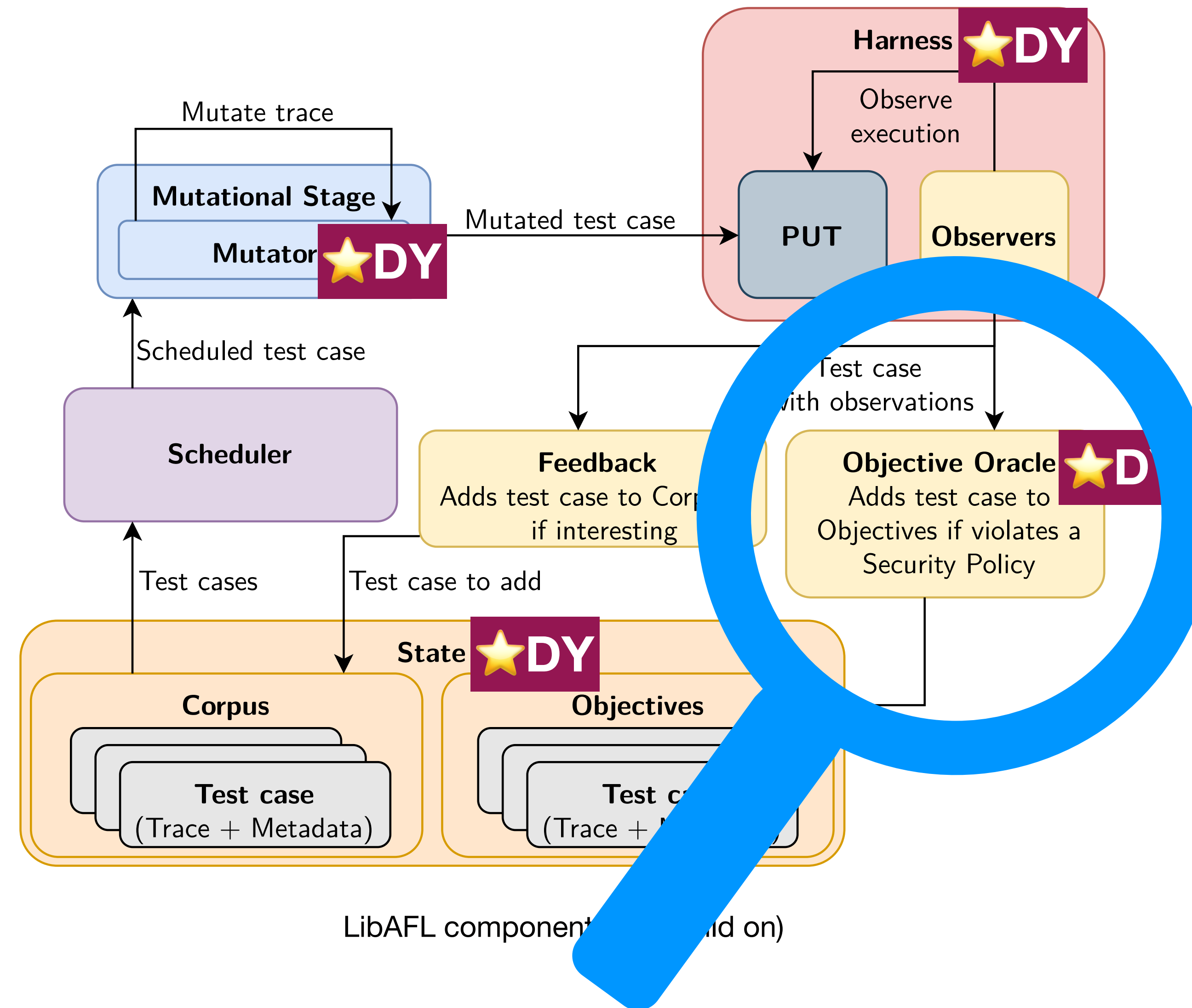
DY Fuzzer components

- **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



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: $\text{Agr}(\text{client}, \text{pk}, \text{m})@i$ client believes to have agreed with server with pk on m @ i^{th} action
- Classical in DY models: security properties expressed as 1st-order formula
E.g. agreement property $\forall \text{pk}, \text{m}: \text{Agr}(\text{client}, \text{pk}, \text{m})@i \Rightarrow \text{Run}(\text{server}, \text{pk}, \text{m})@j \wedge j < i$
- 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 into 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)
- Built on LibAFL, a modular library to build fuzzers (+ new/custom components★)
- In-memory buffers, delightfully parallel, fast (700 execs/s/core)
- Modular: new protocol and new PUTs can be added
- For TLS: 189 function symbols, harnessed PUTs: OpenSSL, WolfSSL, BoringSSL, LibreSSL

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

Once-for-all
6k 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

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

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

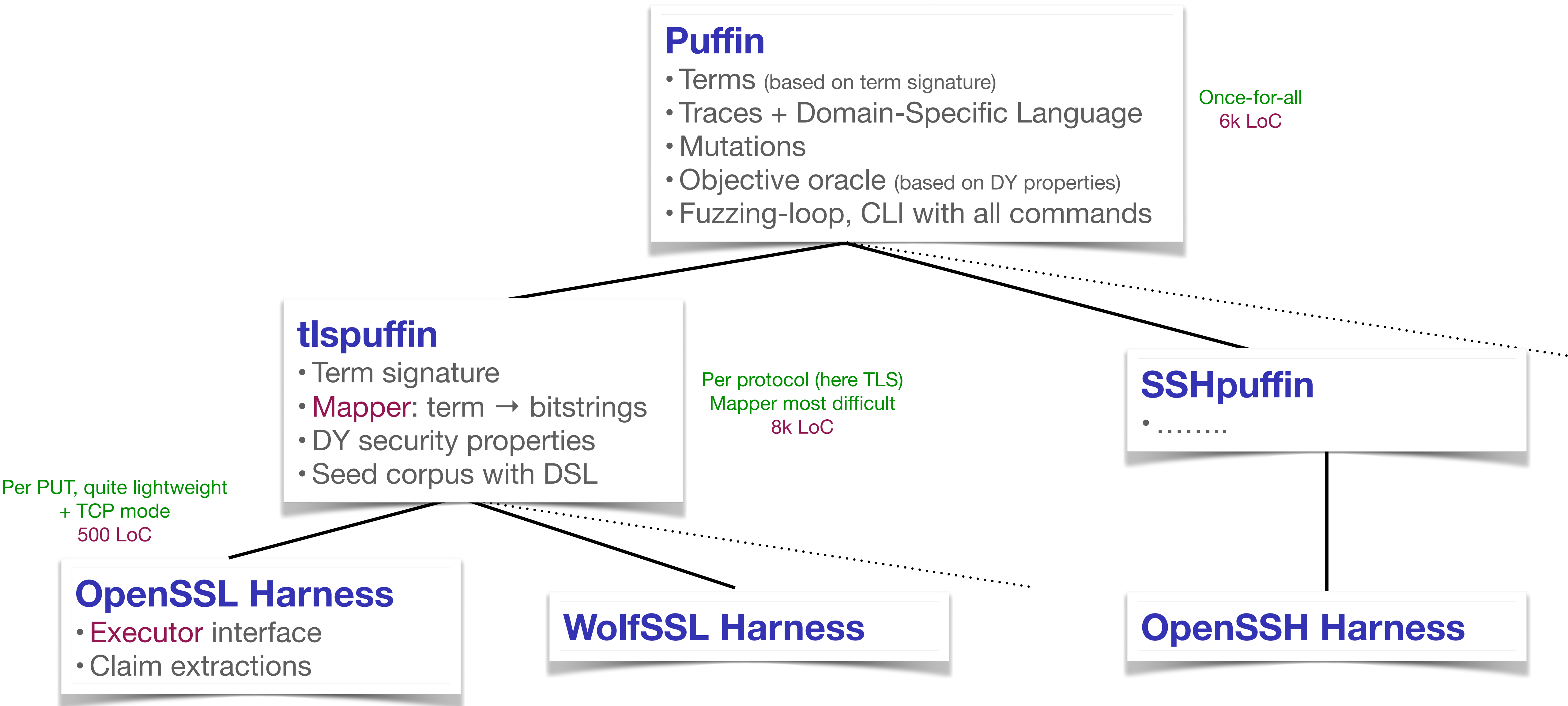
Per PUT, quite lightweight
+ TCP mode
500 LoC

OpenSSL Harness

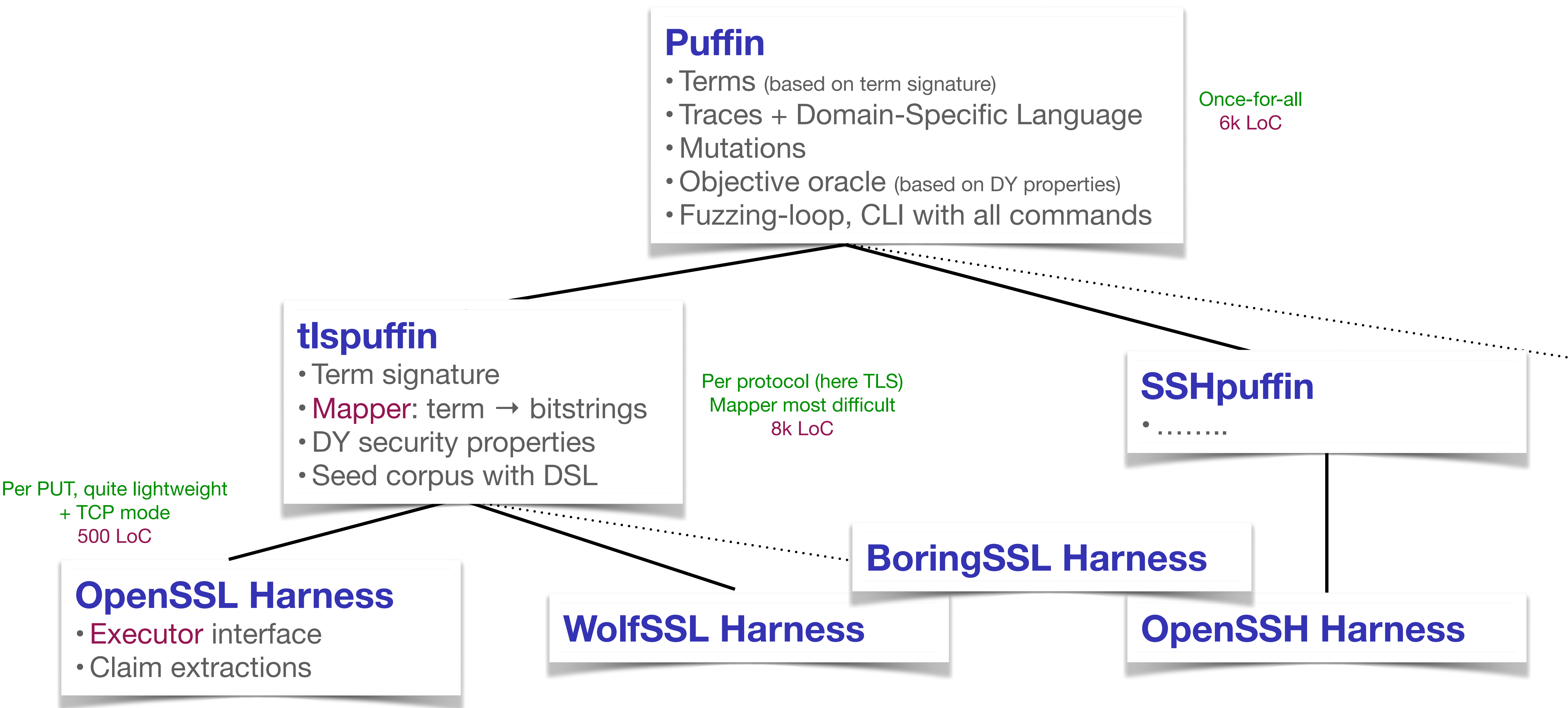
- **Executor** interface
- Claim extractions

WolfSSL Harness

Puffin Modular Architecture



Puffin Modular Architecture



Adding a new protocol

A. Term signature

```
define_signature!(  
  TLS_SIGNATURE,  
  fn_client_hello  
  fn_encrypted_extensions  
  fn_heartbeat  
  fn_hello_retry_request  
  fn_key_update  
  fn_message_hash  
  fn_server_hello  
  fn_server_key_exchange  
  fn_application_data  
  fn_key_share_deterministic_extension  
  // . . .
```

Adding a new protocol

A. Term signature

```
define_signature!(  
  TLS_SIGNATURE,  
  fn_client_hello  
  fn_encrypted_extensions  
  fn_heartbeat  
  fn_hello_retry_request  
  fn_key_update  
  fn_message_hash  
  fn_server_hello  
  fn_server_key_exchange  
  fn_application_data  
  fn_key_share_deterministic_extension  
  // . . .
```

tlspuffin

- A. Term signature
- B. Mapper: term → bitstrings
- C. DY security properties
- D. Seed corpus with DSL

Adding a new protocol

A. Term signature

```
define_signature!(  
  TLS_SIGNATURE,  
  fn_client_hello  
  fn_encrypted_extensions  
  fn_heartbeat  
  fn_hello_retry_request  
  fn_key_update  
  fn_message_hash  
  fn_server_hello  
  fn_server_key_exchange  
  fn_application_data  
  fn_key_share_deterministic_extension  
  // . . .  
)
```

B. Mapper

```
pub fn fn_key_share_deterministic_extension(  
  group: &NamedGroup,  
) -> Result<ClientExtension, FnError> {  
  fn_key_share_extension(&det_key_share(group)?, group)  
}
```

tlspuffin

- A. Term signature
- B. Mapper: term → bitstrings
- C. DY security properties
- D. Seed corpus with DSL

Adding a new protocol

A. Term signature

```
define_signature!(
  TLS_SIGNATURE,
  fn_client_hello
  fn_encrypted_extensions
  fn_heartbeat
  fn_hello_retry_request
  fn_key_update
  fn_message_hash
  fn_server_hello
  fn_server_key_exchange
  fn_application_data
  fn_key_share_deterministic_extension
  // . . .
```

B. Mapper

```
pub fn fn_key_share_deterministic_extension(
  group: &NamedGroup,
) -> Result<ClientExtension, FnError> {
  fn_key_share_extension(&det_key_share(group)?, group)
}
```

C. Security Properties

```
struct TlsClaim {
  pub agent_name: AgentName,
  pub origin: AgentType,
  pub version: TLSVersion,
  pub data: ClaimData,
}
```

```
fn check(claims: &[TlsClaim]) -> Result<(), E> {
  if let Some((client, server)) = find_two(claims) {
    if client.server_random != server.server_random {
      return Err(EP "Mismatching server random");
    }
    // . . .
```

tlspuffin

- A. Term signature
- B. Mapper: term → bitstrings
- C. DY security properties
- D. Seed corpus with DSL

Adding a new protocol

A. Term signature

```
define_signature!(
  TLS_SIGNATURE,
  fn_client_hello
  fn_encrypted_extensions
  fn_heartbeat
  fn_hello_retry_request
  fn_key_update
  fn_message_hash
  fn_server_hello
  fn_server_key_exchange
  fn_application_data
  fn_key_share_deterministic_extension
  // . . .
```

B. Mapper

```
pub fn fn_key_share_deterministic_extension(
  group: &NamedGroup,
) -> Result<ClientExtension, FnError> {
  fn_key_share_extension(&det_key_share(group)?, group)
}
```

C. Security Properties

```
struct TlsClaim {
  pub agent_name: AgentName,
  pub origin: AgentType,
  pub version: TLSVersion,
  pub data: ClaimData,
}
```

```
fn check(claims: &[TlsClaim]) -> Result<(), E> {
  if let Some((client, server)) = find_two(claims) {
    if client.server_random != server.server_random {
      return Err(EP "Mismatching server random");
    }
    // . . .
  }
```

D. Seed Corpus

```
pub fn seed_successful(client: AgentName, server: AgentName) -> Trace {
  Trace {
    steps: vec![
      OutputAction::new_step(client),
      // Client Hello Client -> Server
      Step {
        agent: server,
        action: Action::Input(InputAction {
          recipe: term! {
            (client, 0)/MessageFlight
          },
        }),
      },
      // ServerHello/.../ServerFinished -> Client
      Step {
        agent: client,
        action: Action::Input(InputAction {
          recipe: term! {
            (server, 0)/MessageFlight
          },
        }),
      },
      // . . .
    ],
  }
```

tlspuffin

- A. Term signature
- B. Mapper: term $\rightarrow$ bitstrings
- C. DY security properties
- D. Seed corpus with DSL

Adding a new PUT

Expose a C or Rust interface:

Adding a new PUT

Expose a C or Rust interface:

```
typedef struct C_PUT_INTERFACE  
{
```

Adding a new PUT

Expose a C or Rust interface:

```
typedef struct C_PUT_INTERFACE
{
    // Agent handling
    void *(*const create)(const AGENT_DESCRIPTOR *descriptor);
```

Adding a new PUT

Expose a C or Rust interface:

```
typedef struct C_PUT_INTERFACE
{
    // Agent handling
    void *(*const create)(const AGENT_DESCRIPTOR *descriptor);
    RESULT (*const progress)(void *agent);
    RESULT (*const reset)(void *agent);
}
```

Adding a new PUT

Expose a C or Rust interface:

```
typedef struct C_PUT_INTERFACE
{
    // Agent handling
    void *(*const create)(const AGENT_DESCRIPTOR *descriptor);
    RESULT (*const progress)(void *agent);
    RESULT (*const reset)(void *agent);

    // Claims handling
    void (*const register_claimer)(void *agent, const CLAIMER_CB *callback);
}
```

Adding a new PUT

Expose a C or Rust interface:

```
typedef struct C_PUT_INTERFACE
{
    // Agent handling
    void *(*const create)(const AGENT_DESCRIPTOR *descriptor);
    RESULT (*const progress)(void *agent);
    RESULT (*const reset)(void *agent);

    // Claims handling
    void (*const register_claimer)(void *agent, const CLAIMER_CB *callback);

    // Input
    RESULT (*const add_inbound)(void *agent, const uint8_t *bytes, size_t length, size_t *written);
}
```

Adding a new PUT

Expose a C or Rust interface:

```
typedef struct C_PUT_INTERFACE
{
    // Agent handling
    void *(*const create)(const AGENT_DESCRIPTOR *descriptor);
    RESULT (*const progress)(void *agent);
    RESULT (*const reset)(void *agent);

    // Claims handling
    void (*const register_claimer)(void *agent, const CLAIMER_CB *callback);

    // Input
    RESULT (*const add_inbound)(void *agent, const uint8_t *bytes, size_t length, size_t *written);

    // Output
    RESULT (*const take_outbound)(void *agent, uint8_t *bytes, size_t max_length, size_t *readbytes);
    // ...
} C_PUT_INTERFACE;
```

tlspuffin Results

tlspuffin findings 🤡

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 **5 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

+ CVE-2023-6936 on WolfSSL, under discussion

Short demo

Short demo

- Install and test tlspuffin: <https://tlspuffin.github.io/docs/guides/quickstart>
- Run a fuzzing campaign against WolfSSL v and find an objective
- Re-test the objective is legit (see also <https://tlspuffin.github.io/docs/guides/getting-started/replaying-traces>)
- Inspect the trace underlying the found objective

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: $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$

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: $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$

How WolfSSL implements $\cap$ 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  $\cap$  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: $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$

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:** $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$
- (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:** $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$
 - (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)

An overflow on the stack of max 44700 bytes (controlled by `n` so is attacker 🤡-controlled).

- 👉 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)

(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`

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** and **prone to exhaustion**

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 $\sim_{coverage}$ 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** and **prone to exhaustion**

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 $\sim_{coverage}$ accepting illegitimate cert.

👉 Need for a domain-specific DY-based notion of coverage + balance with code-cov.

Future Work (cont.)

Future Work (cont.)

Improved objective oracle

- **Differential fuzzing:** save t as objective when $\text{WolfSSL}(t) \neq \text{OpenSSL}(t)$ [WIP Tom]
- **Or extend the oracle:** +properties & +compromise scenarios

Future Work (cont.)

Improved objective oracle

- **Differential fuzzing:** save t as objective when $\text{WolfSSL}(t) \neq \text{OpenSSL}(t)$ [WIP Tom]
- **Or extend the oracle:** +properties & +compromise scenarios

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

Future Work (cont.)

Improved objective oracle

- **Differential fuzzing:** save t as objective when $\text{WolfSSL}(t) \neq \text{OpenSSL}(t)$ [WIP Tom]
- **Or extend the oracle:** +properties & +compromise scenarios

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

Apply DY fuzzing to more protocols (e.g. WPA2, TelCo) and PUTs [WIP Lucca]

Easing new PUT/protocol integration [WIP Michael]

Future Work (cont.)

Improved objective oracle

- **Differential fuzzing**: save t as objective when $\text{WolfSSL}(t) \neq \text{OpenSSL}(t)$ [WIP Tom]
- **Or extend the oracle**: +properties & +compromise scenarios

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

Apply DY fuzzing to more protocols (e.g. WPA2, TelCo) and PUTs [WIP Lucca]

Easing new PUT/protocol integration [WIP Michael]

Long-Term :

- (Partially) **Automate** Mapper and Harness → **PUT-agnostic DY fuzzer**
- Connect further with **DY verifiers** (ProVerif, Tamarin)



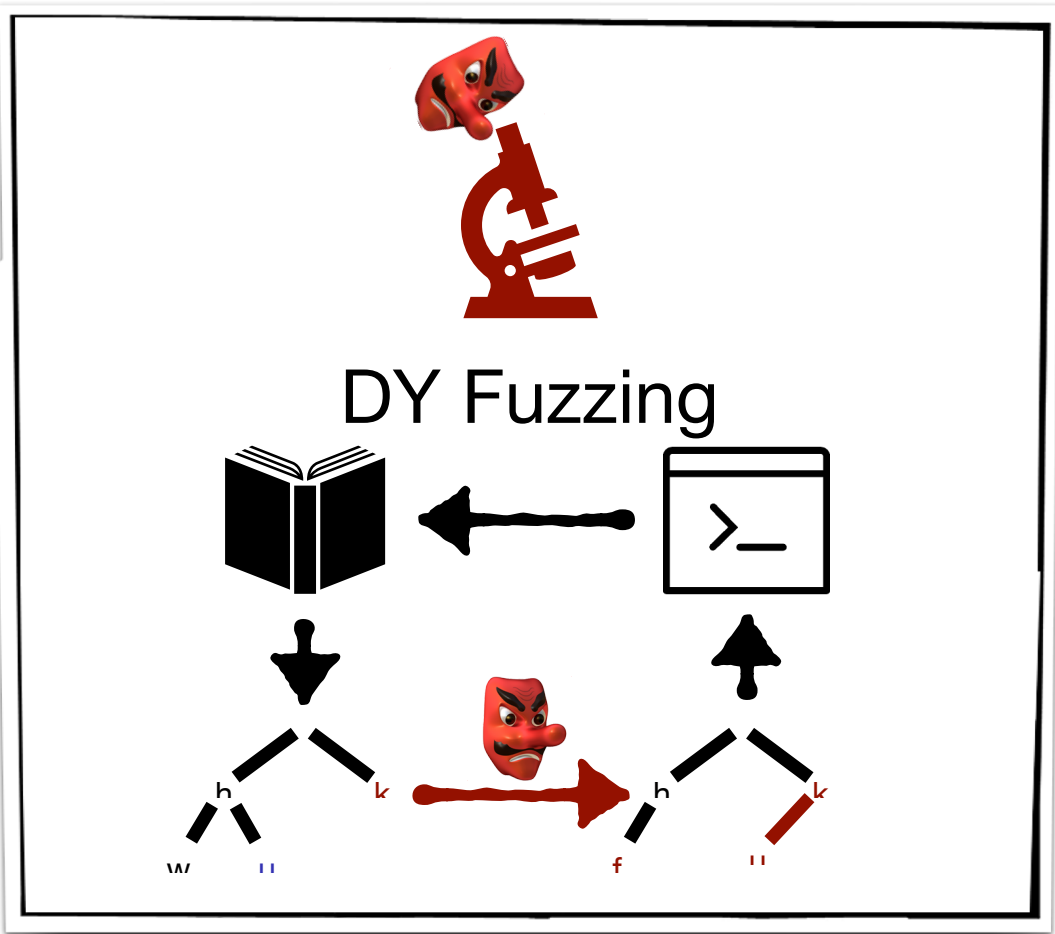
DY Fuzzer = DY attacker in a fuzzing loop



Checkout our website:
<https://tlspuffin.github.io>

We propose a **new fuzzer design** (DY fuzzing)

- ✓ New notion of **executable DY test-cases** based on formal terms
- ✓ New **DY mutations** capturing all **DY attacker** 🦹's actions
- ✓ New **bug oracle** to detect **protocol vulnerabilities**



We propose **tlspuffin**: DY fuzzer implementation

- ✓ **Open-source**, in Rust, built on top of **LibAFL**
- ✓ **Modular**: new protocols (beyond TLS) and PUTs can be added
- ✓ **Current Protocol Implementations**: Open/Wolf/Libre/BoringSSL, OpenSSH (WIP)
- ✓ Bench + Evaluation + 8 CVEs (5 new ones, acknowledged and patched)

(New)

CVE ID	CVSS	Type	New?	Target
2021-3449	5.9	Server DoS, M	✗	OpenSSL
2022-25638	6.5	Auth. Bypass, P	✗	WolfSSL
2022-25640	7.5	Auth. Bypass, P	✗	WolfSSL
2022-38152	7.5	Client DoS, M	✓	WolfSSL
2022-38153	5.9	Server DoS, M	✓	WolfSSL
2022-39173	7.5	Server DoS, M	✓	WolfSSL
2022-42905	9.1	Info. Leak, M	✓	WolfSSL
2023-6936	5.3	Info. Leak, M	✓	WolfSSL

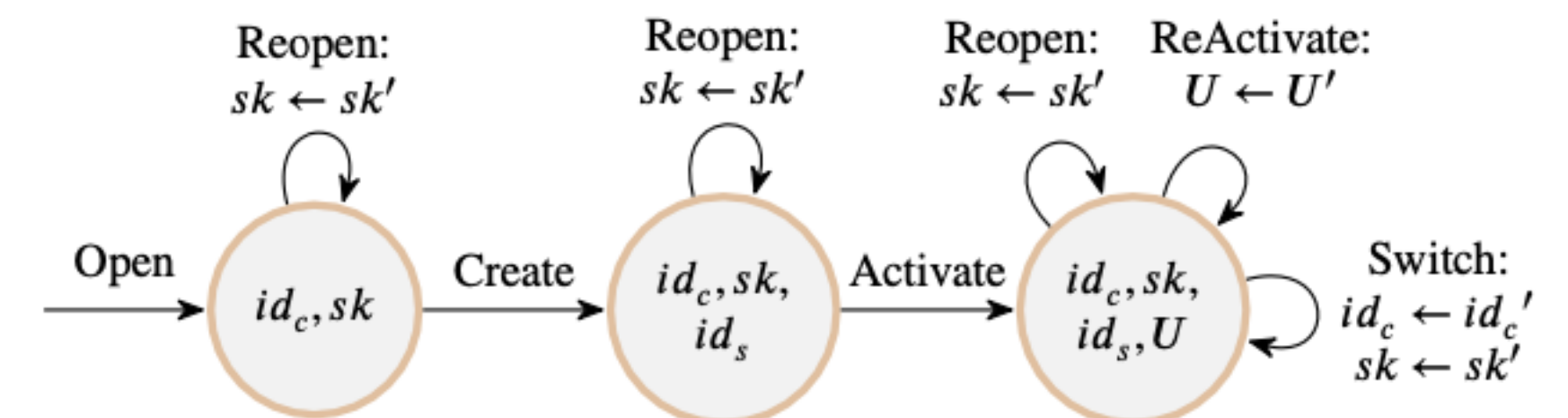
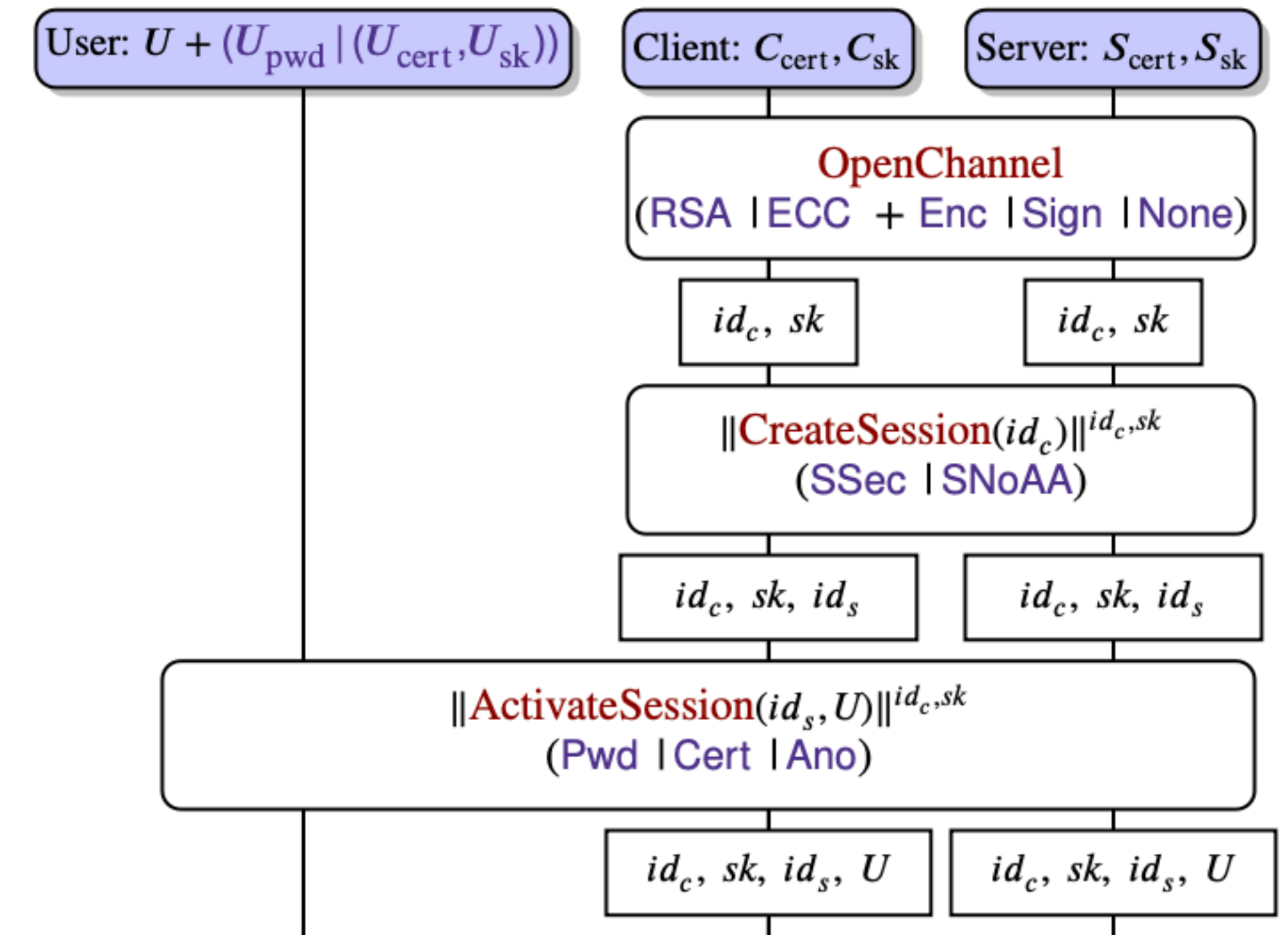
A Comprehensive Formal Security Analysis of OPC-UA

[Vincent Diemunsch](#), Inria & ANSSI, France

[Lucca Hirschi](#), Inria, France

[Steve Kremer](#), Inria, France

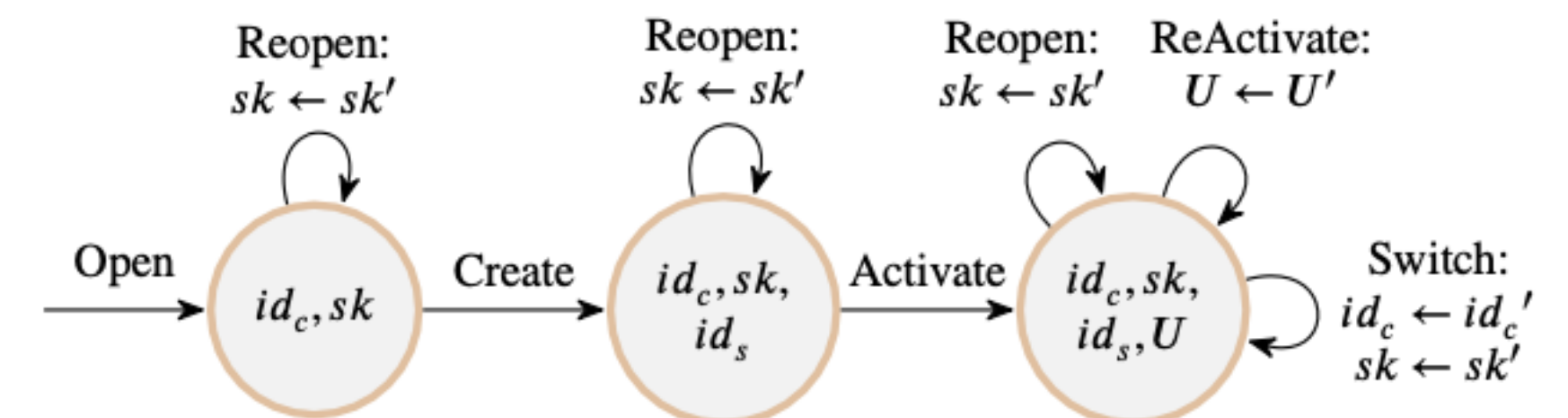
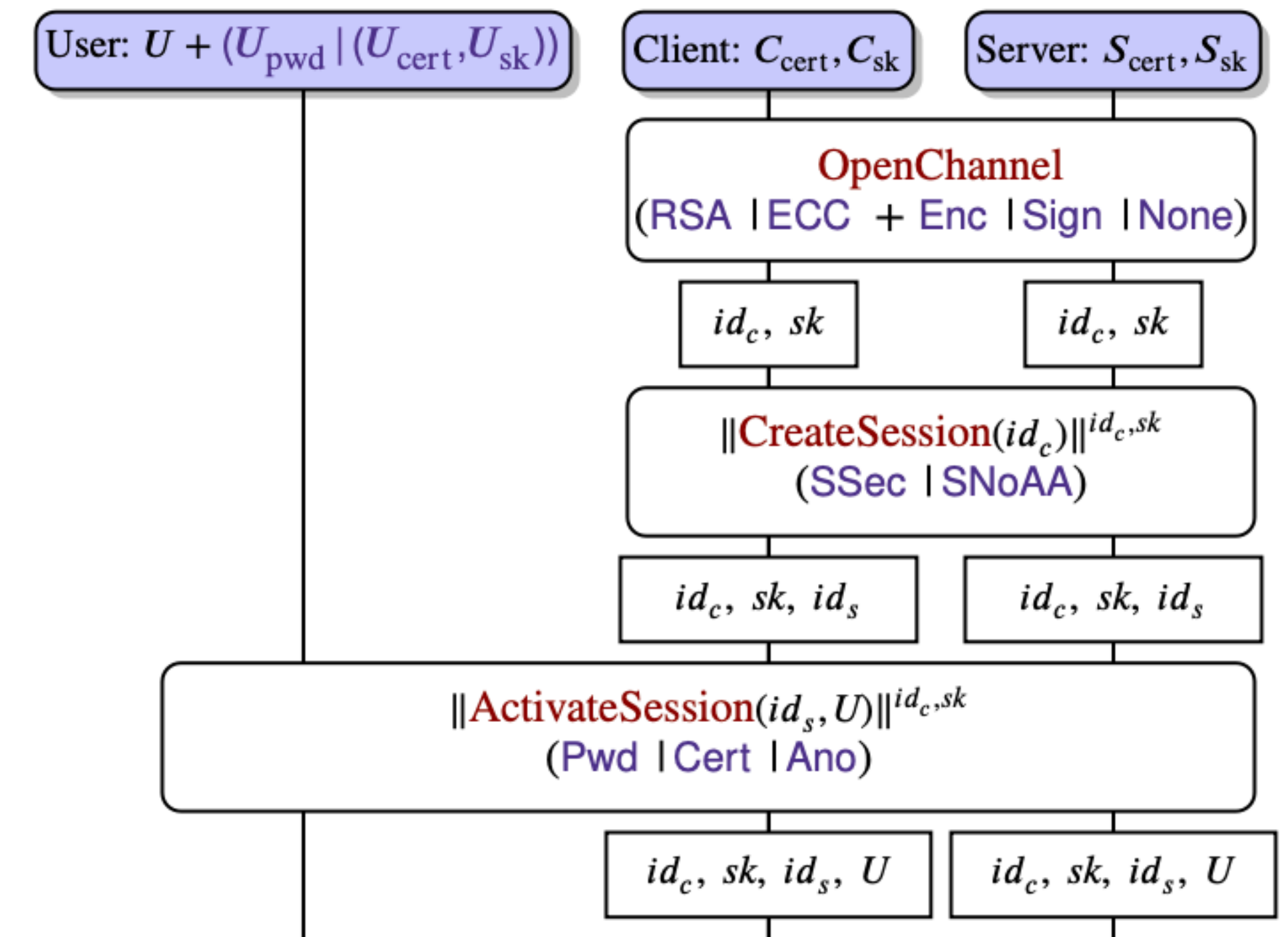
OPC-UA: a general purpose ICS protocol



OPC-UA: a general purpose ICS protocol

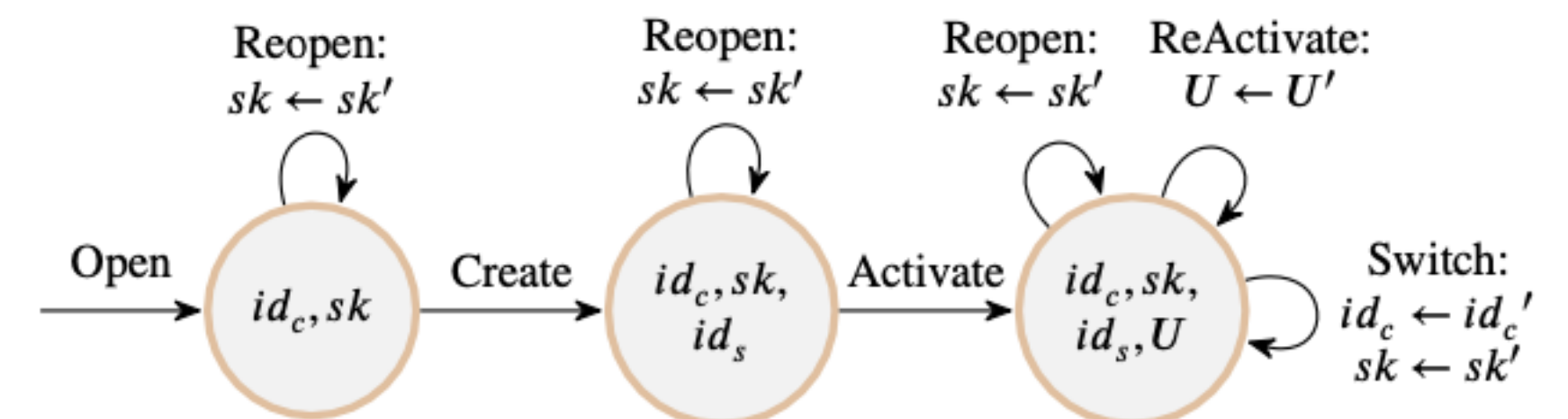
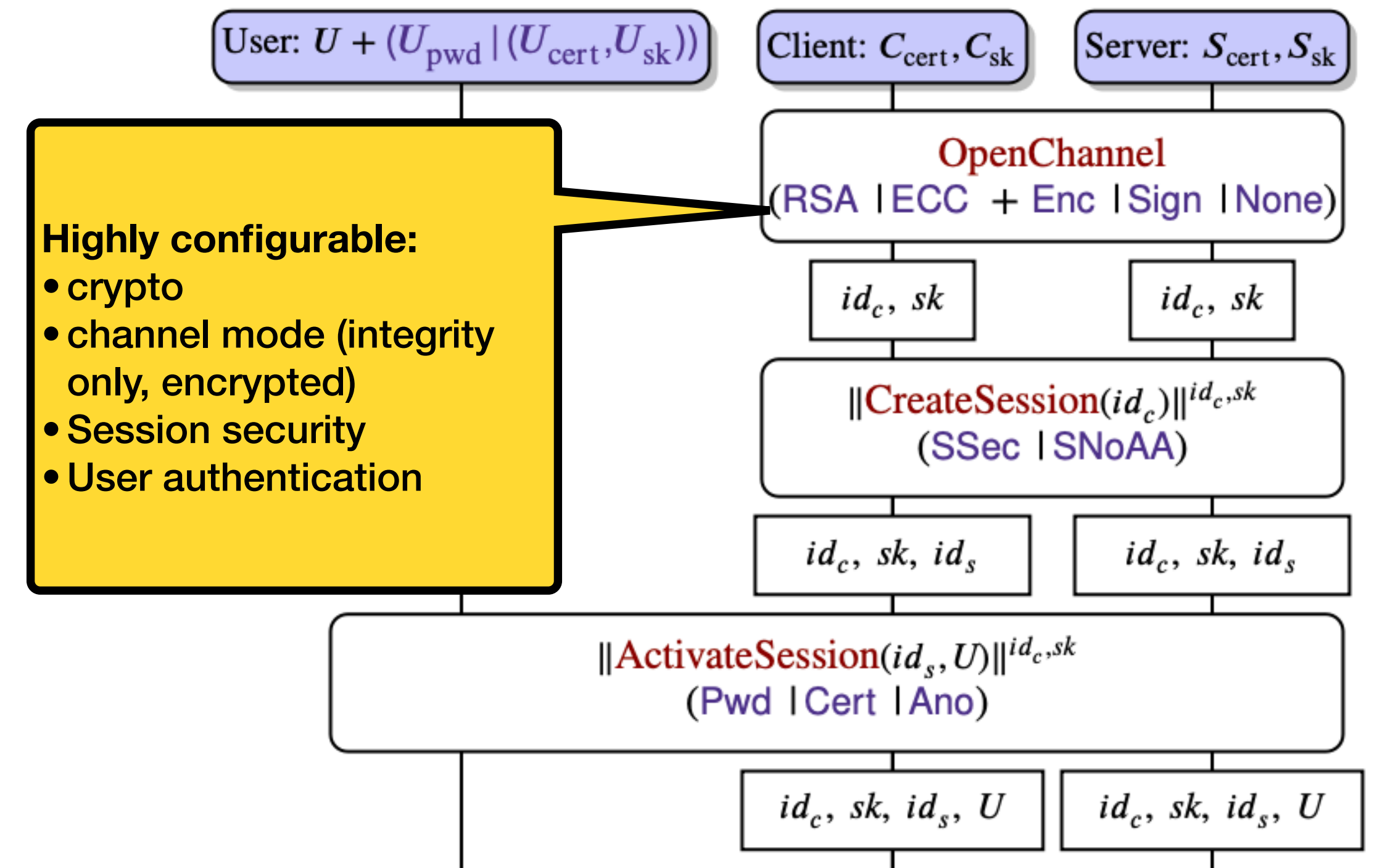
- Protocol for Industrial Control Systems
- Constraints from OT (safety and reliability)
- OPC-UA: security built-in
- standardised (v1.04) as IEC 62541
- new version (1.05) adds ECDH (formerly RSA only)

- more efficient
- Perfect Forward Secrecy



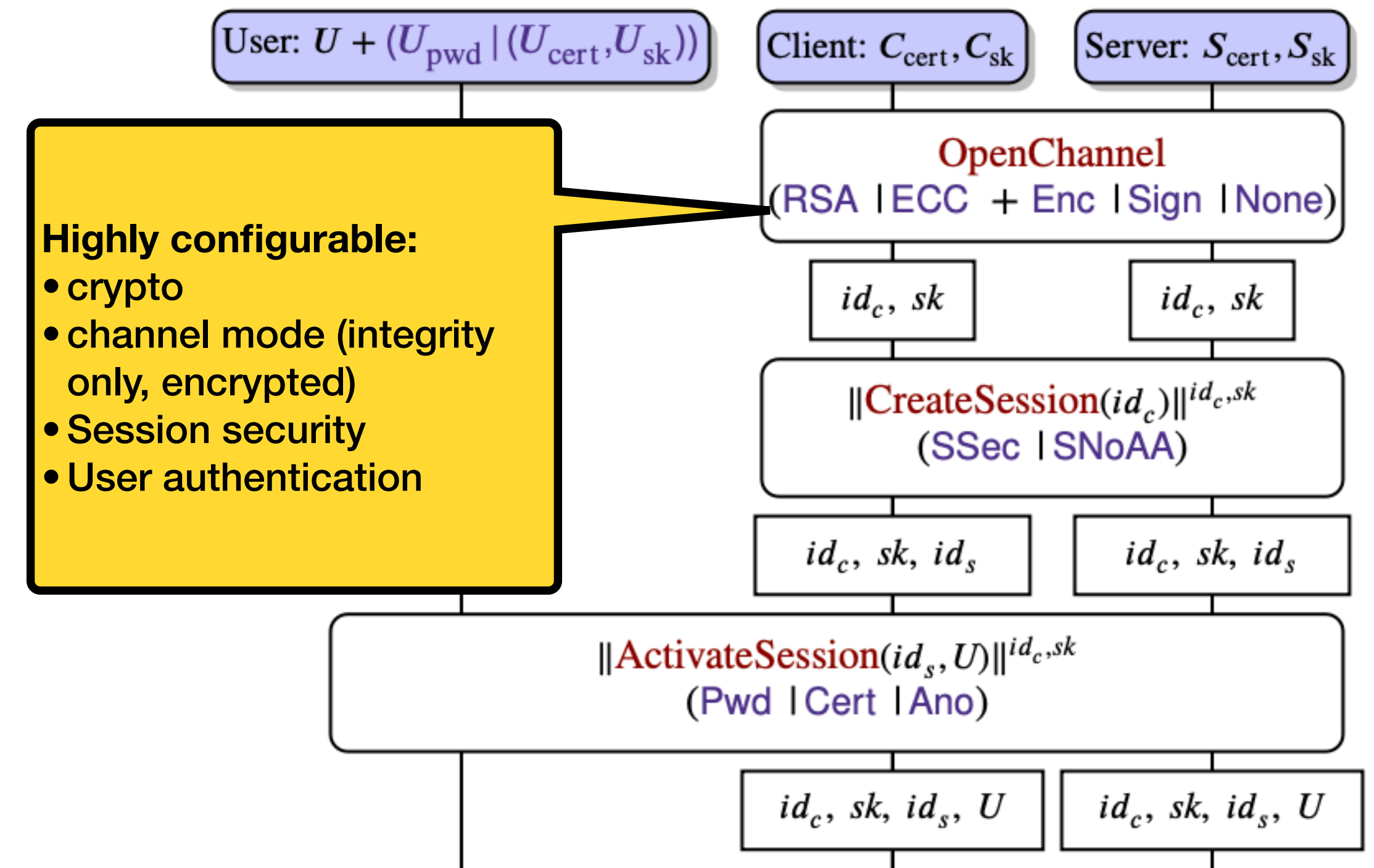
OPC-UA: a general purpose ICS protocol

- Protocol for Industrial Control Systems
 - Constraints from OT (safety and reliability)
 - OPC-UA: security built-in
 - standardised (v1.04) as IEC 62541
 - new version (1.05) adds ECDH (formerly RSA only)
- more efficient
 - Perfect Forward Secrecy



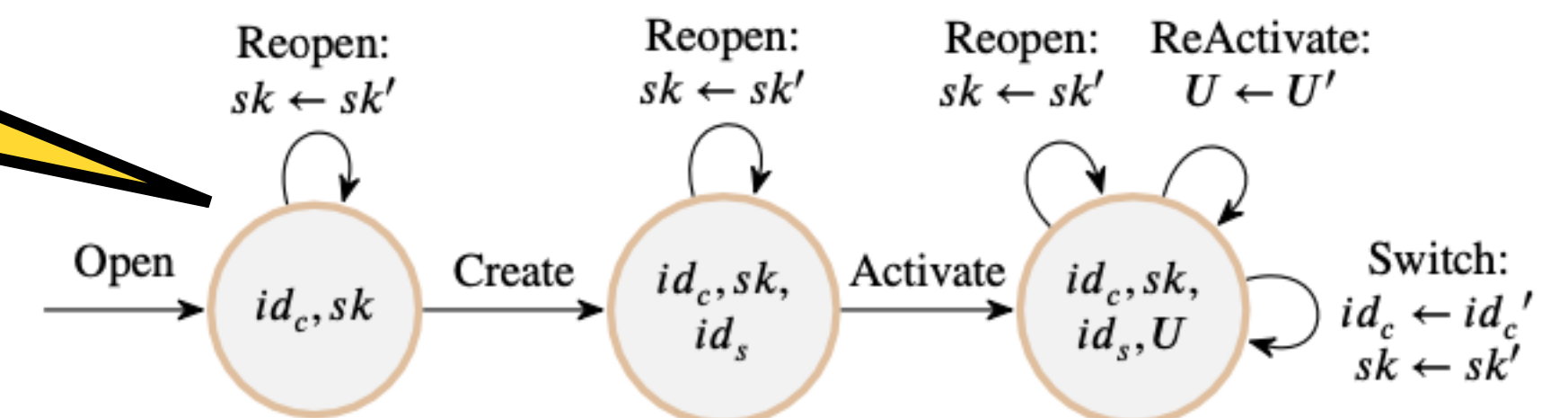
OPC-UA: a general purpose ICS protocol

- Protocol for Industrial Control Systems
- Constraints from OT (safety and reliability)
- OPC-UA: security built-in
- standardised (v1.04) as IEC 62541
- new version (1.05) adds ECDH (formerly RSA only)



- more efficient
- Perfect Forward Secrecy

Complex state machine



Formal analysis using ProVerif

Configuration option	Admissible values	# configurations
SecurityPolicy	(ECC RSA)	3
Mode	(Enc Sign None)	7
SessionSecurity	(SSec SNoAA)	3
UserAuthentication	(Cert Pwd Ano)	7
Leak	(Ltk Chk / Nok)	4
Reopen	(T F)	2
Switch	(T F)	2

Properties	OPC-UA v1.05.03	OPC-UA v1.05.04 RC
<i>Conf[C]</i>	✗ 5.6	✓
<i>Conf[S]</i>	✓	✓
<i>Conf[Pwd]</i>	✗ 5.5, 5.6	✓
<i>Agr[S↔C]</i>	✗ 5.2, 5.3, 5.4	✗/✓ -
<i>Agr[C↔S]</i>	✗ 5.2, 5.3, 5.6	✗/✓ -

Formal analysis using ProVerif

- Comprehensive model combining sub-protocols
- Highly configurable + different threat models: 7 056 possible configurations organised in a lattice
 - ➡ largest ProVerif model ever analysed
- Lots of ProVerif wizardry (lemmas, name contexts, selection function, ...) and scripting
- 5 new vulnerabilities/weaknesses (reported, acknowledged and fixed/mitigated by OPC)

Configuration option	Admissible values	# configurations
SecurityPolicy	(ECC RSA)	3
Mode	(Enc Sign None)	7
SessionSecurity	(SSec SNoAA)	3
UserAuthentication	(Cert Pwd Ano)	7
Leak	(Ltk Chk / Nok)	4
Reopen	(T F)	2
Switch	(T F)	2

Properties	OPC-UA v1.05.03	OPC-UA v1.05.04 RC
Conf[C]	✗ 5.6	✓
Conf[S]	✓	✓
Conf[Pwd]	✗ 5.5, 5.6	✓
Agr[S↔C]	✗ 5.2, 5.3, 5.4	✗/✓-
Agr[C↔S]	✗ 5.2, 5.3, 5.6	✗/✓-

Formal analysis using ProVerif

- Comprehensive model combining sub-protocols
- Highly configurable + different threat models: 7 056 possible configurations organised in a lattice
 - ➡ largest ProVerif model ever analysed
- Lots of ProVerif wizardry (lemmas, name contexts, selection function, ...) and scripting
- 5 new vulnerabilities/weaknesses (reported, acknowledged and fixed/mitigated by OPC)

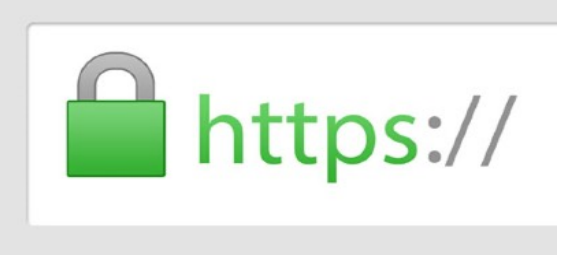
Configuration option	Admissible values	# configurations
SecurityPolicy	(ECC RSA)	3
Mode	(Enc Sign None)	7
SessionSecurity	(SSec SNoAA)	3
UserAuthentication	(Cert Pwd Ano)	7
Leak	(Ltk Chk / Nok)	4
Reopen	(T F)	2
Switch	(T F)	2

Properties	OPC-UA v1.05.03	OPC-UA v1.05.04 RC
Conf[C]	✗ 5.6	✓
Conf[S]	✓	✓
Conf[Pwd]	✗ 5.5, 5.6	✓
Agr[S↔C]	✗ 5.2, 5.3, 5.4	✗/✓-
Agr[C↔S]	✗ 5.2, 5.3, 5.6	✗/✓-

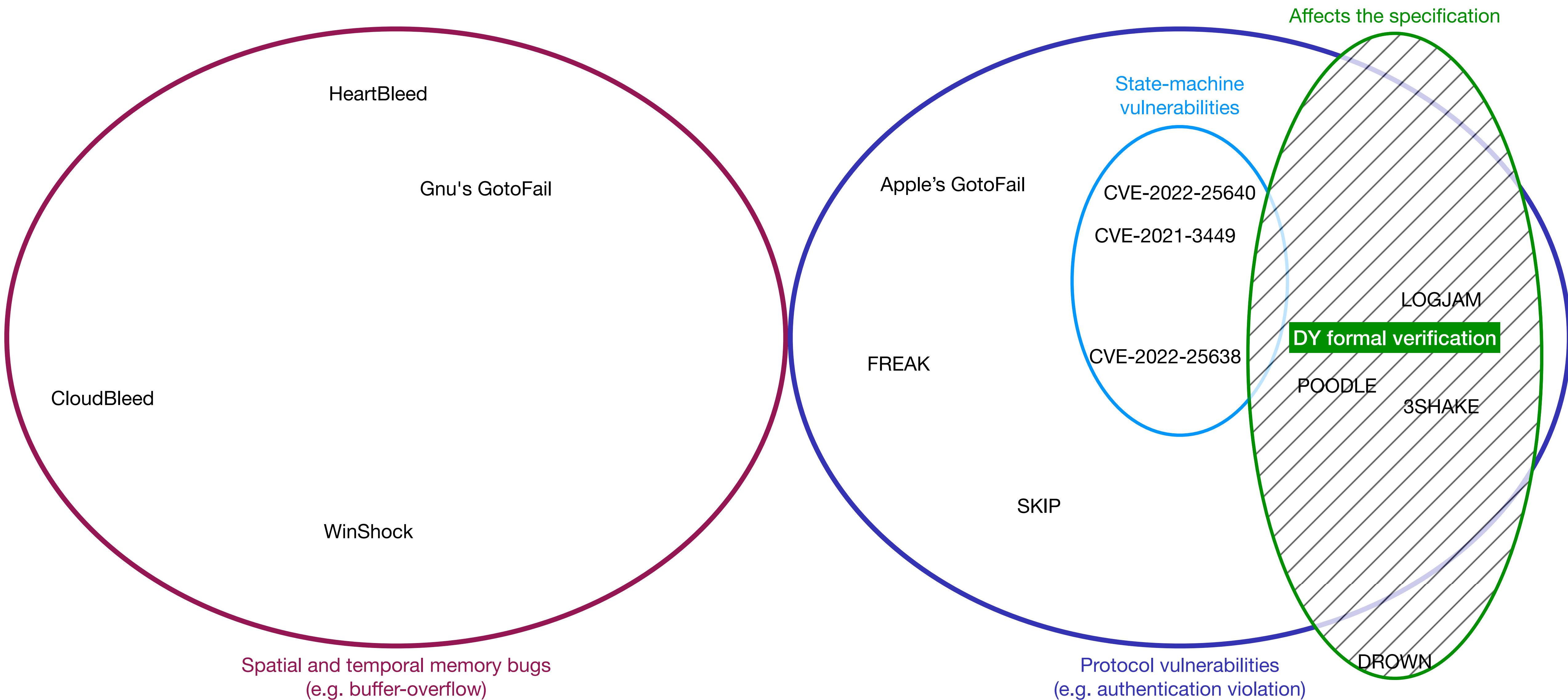
More details in the paper. Under submission, but gladly provided upon request!

Backup Slides

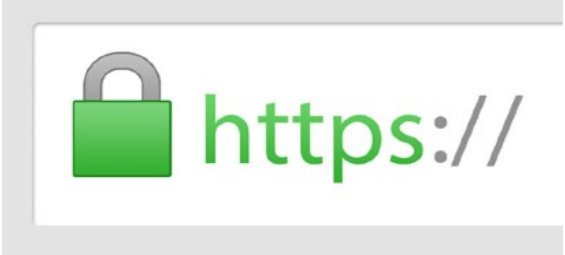
Retrospective of TLS Failures



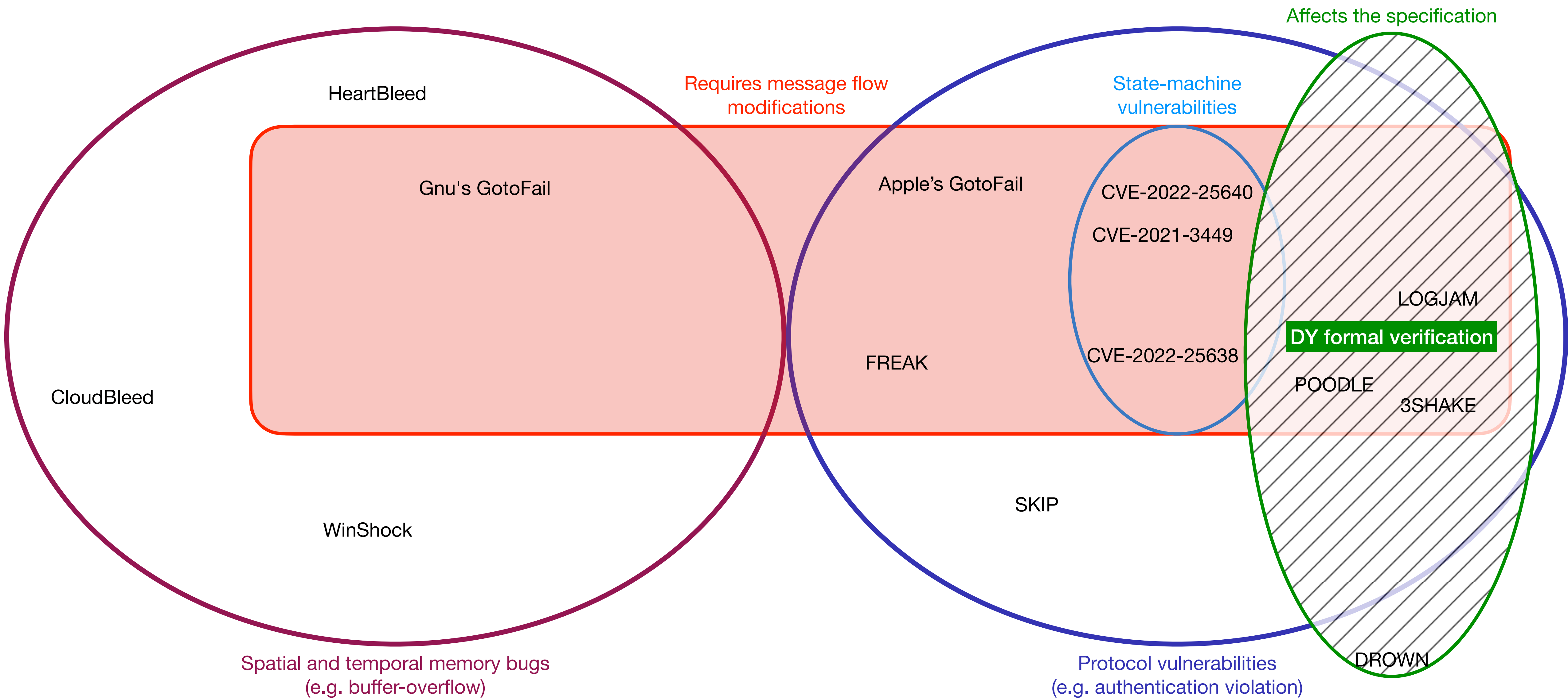
2014-2022



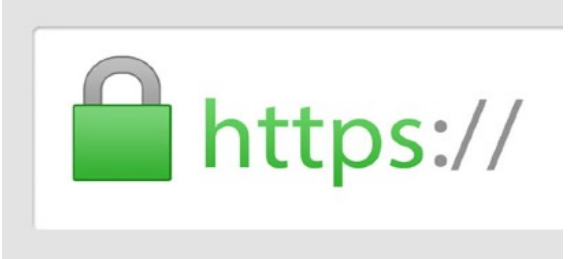
Retrospective of TLS Failures



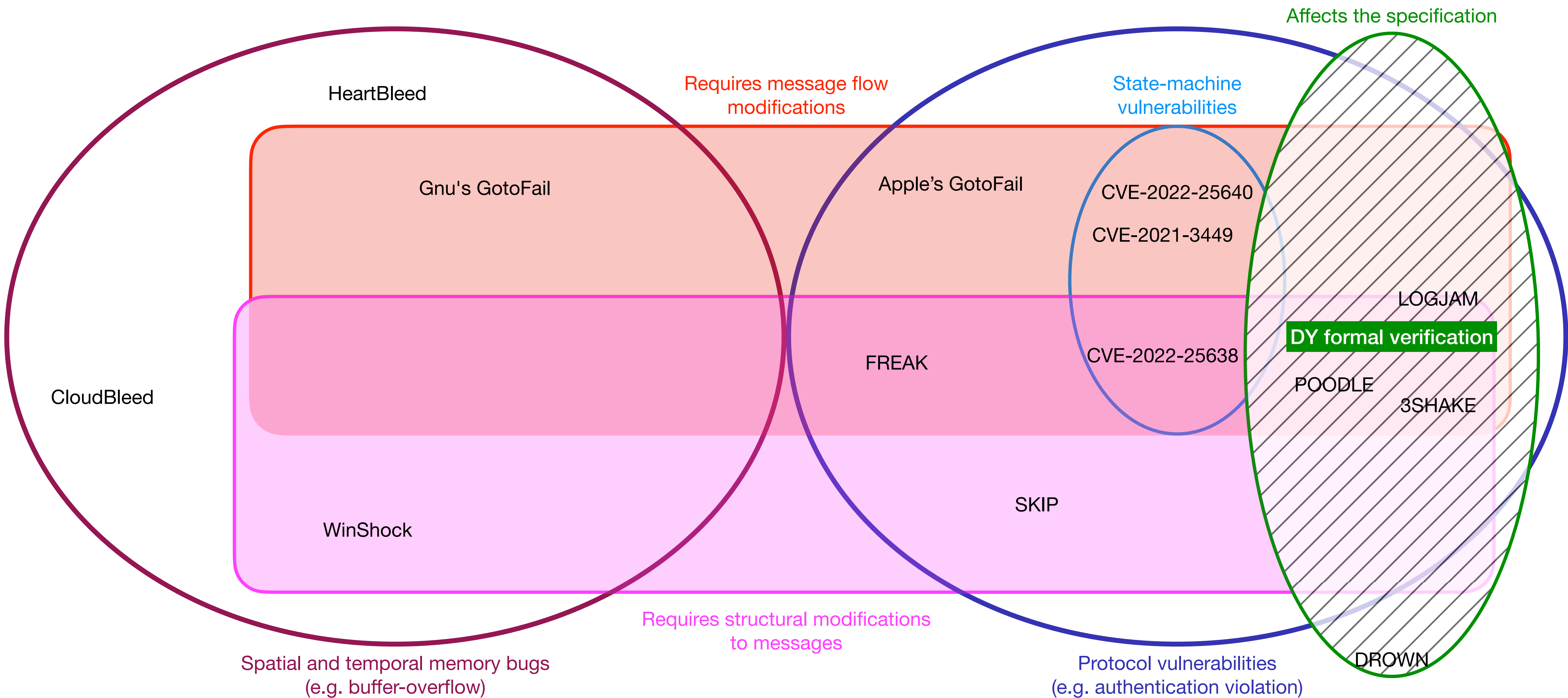
2014-2022



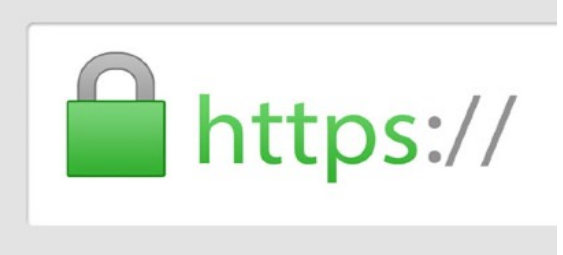
Retrospective of TLS Failures



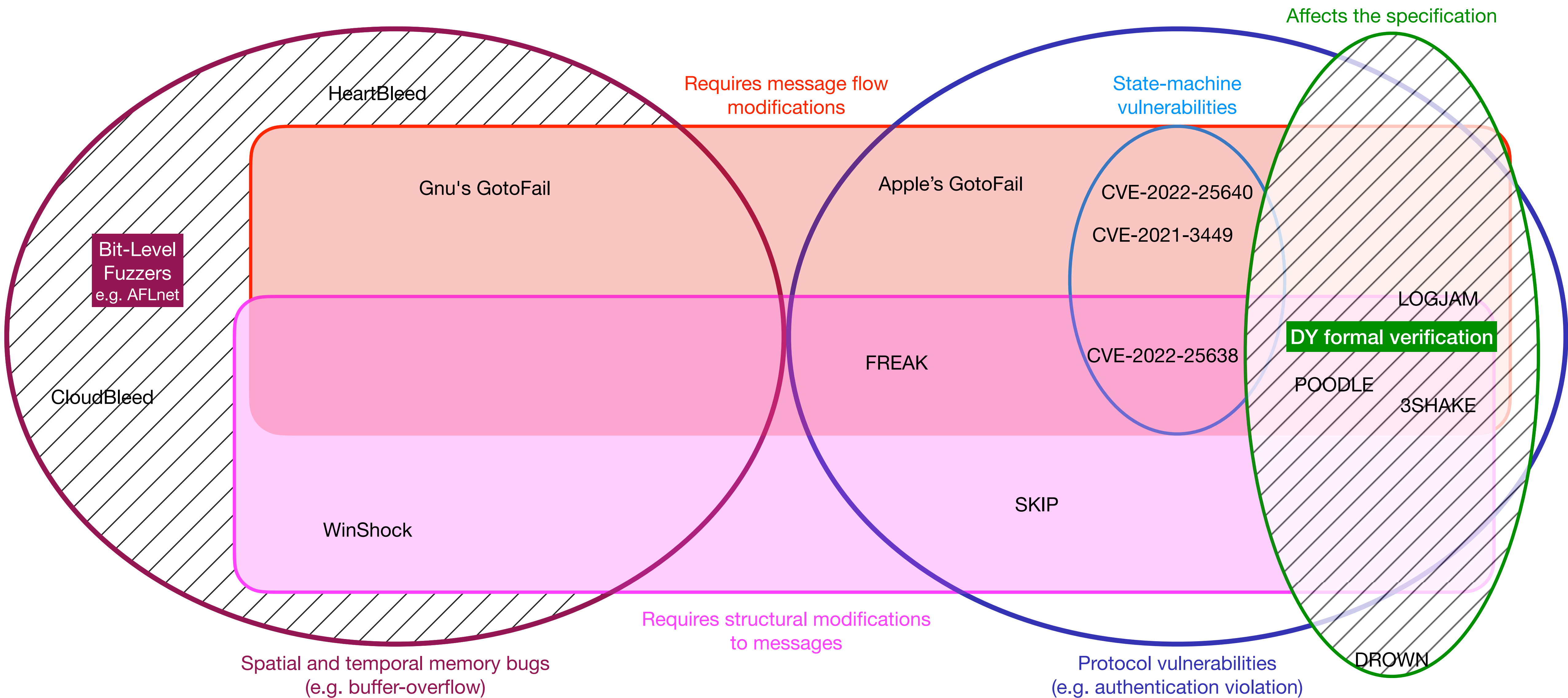
2014-2022



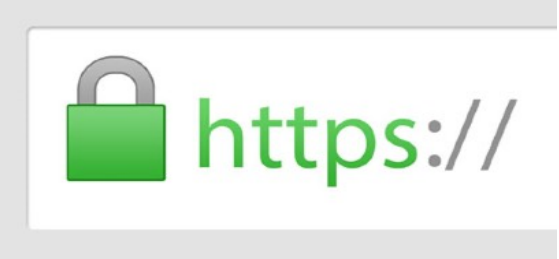
Retrospective of TLS Failures



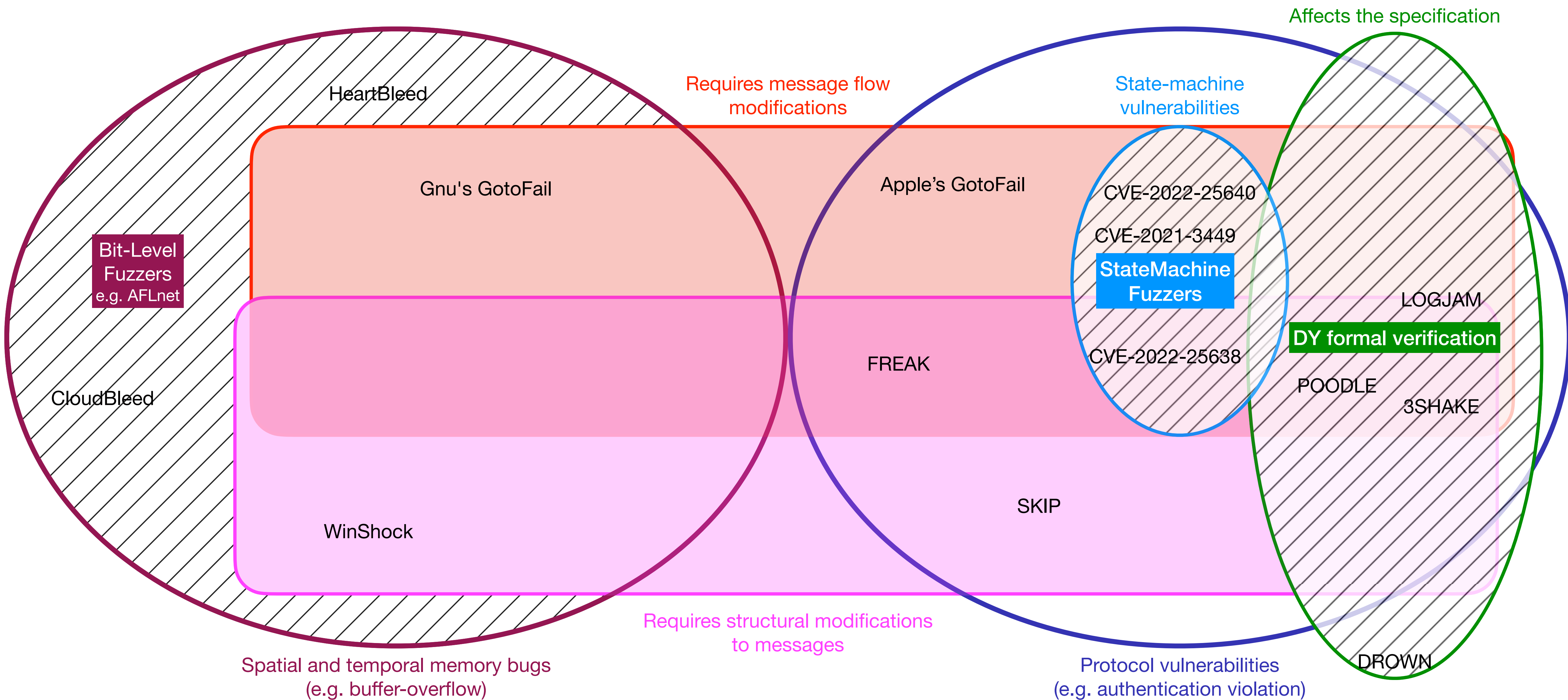
2014-2022



Retrospective of TLS Failures



2014-2022



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/lldb **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: $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$

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: $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$

How WolfSSL implements $\cap$ 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  $\cap$  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: $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$

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:** $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$
 - (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:** $k = n \leq |\text{suitesC}| \leq \text{MAX_SZ} = 150$
 - (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)

An overflow on the stack of max 44700 bytes (controlled by n so is attacker 🤡-controlled).

- 👉 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)

(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

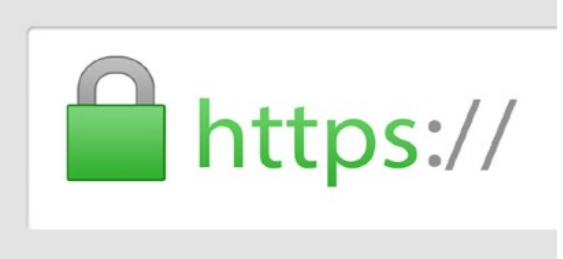
👉 **Flaw 2: side-effects** of `refineSuites` **are not reverted**

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

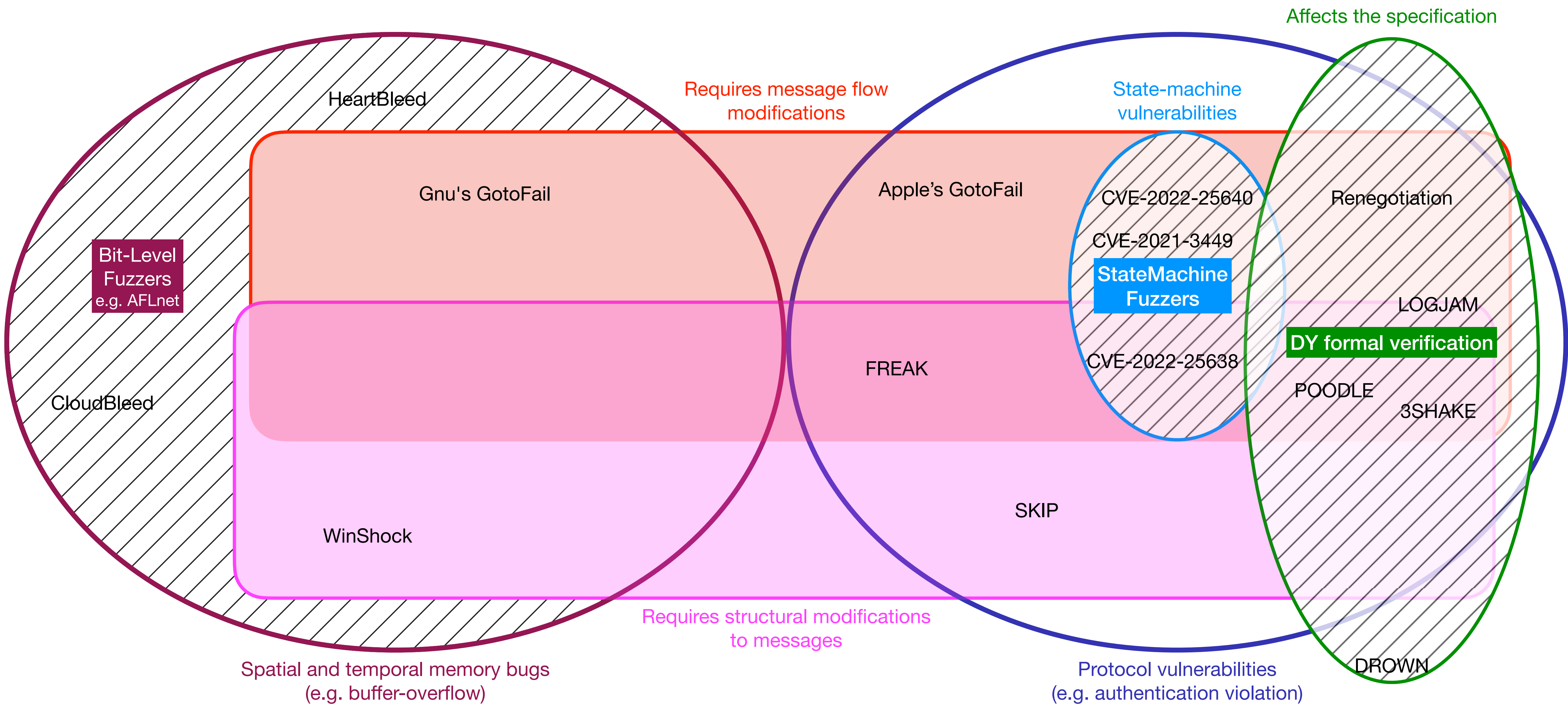
3. Send `ClientHello([c;..;c])` again, `refineSuites` is called again, the resulting buffer `suites` that contains $k^2 = n^2$ ciphers `c` is copied into `suitesS`

👉 For $n = 13$, we already overwrite the `suitesS` buffer allocated on `MAX_ciphers_list_length = 150`

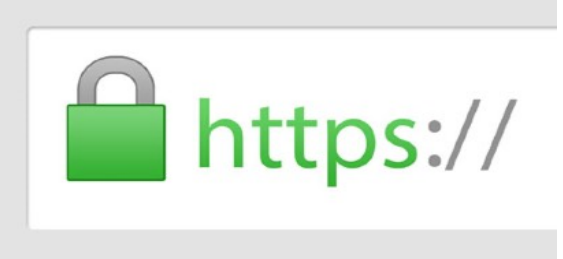
Retrospective of TLS Failures



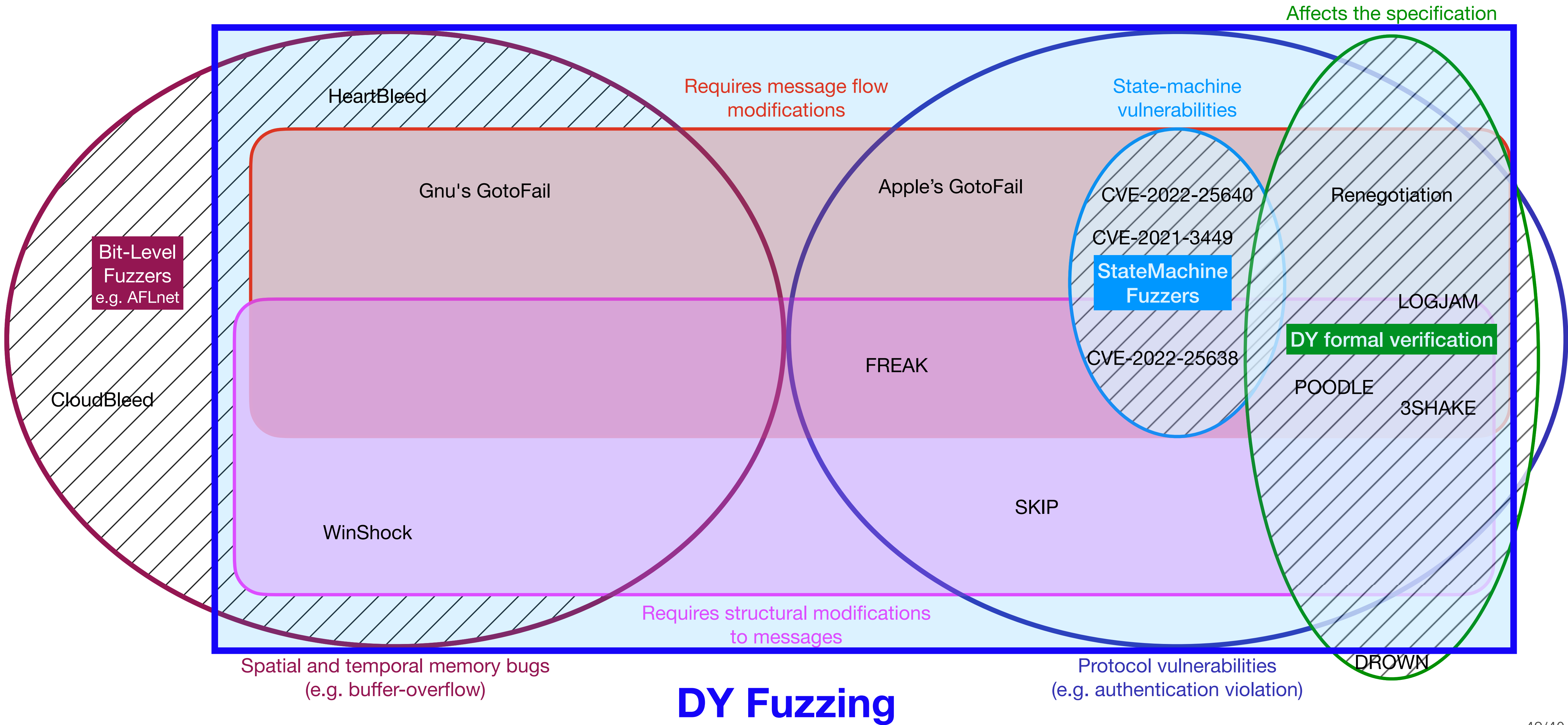
2019-2022



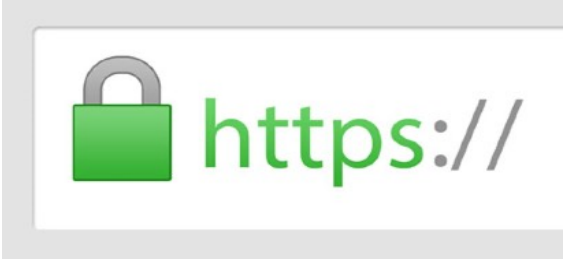
Retrospective of TLS Failures



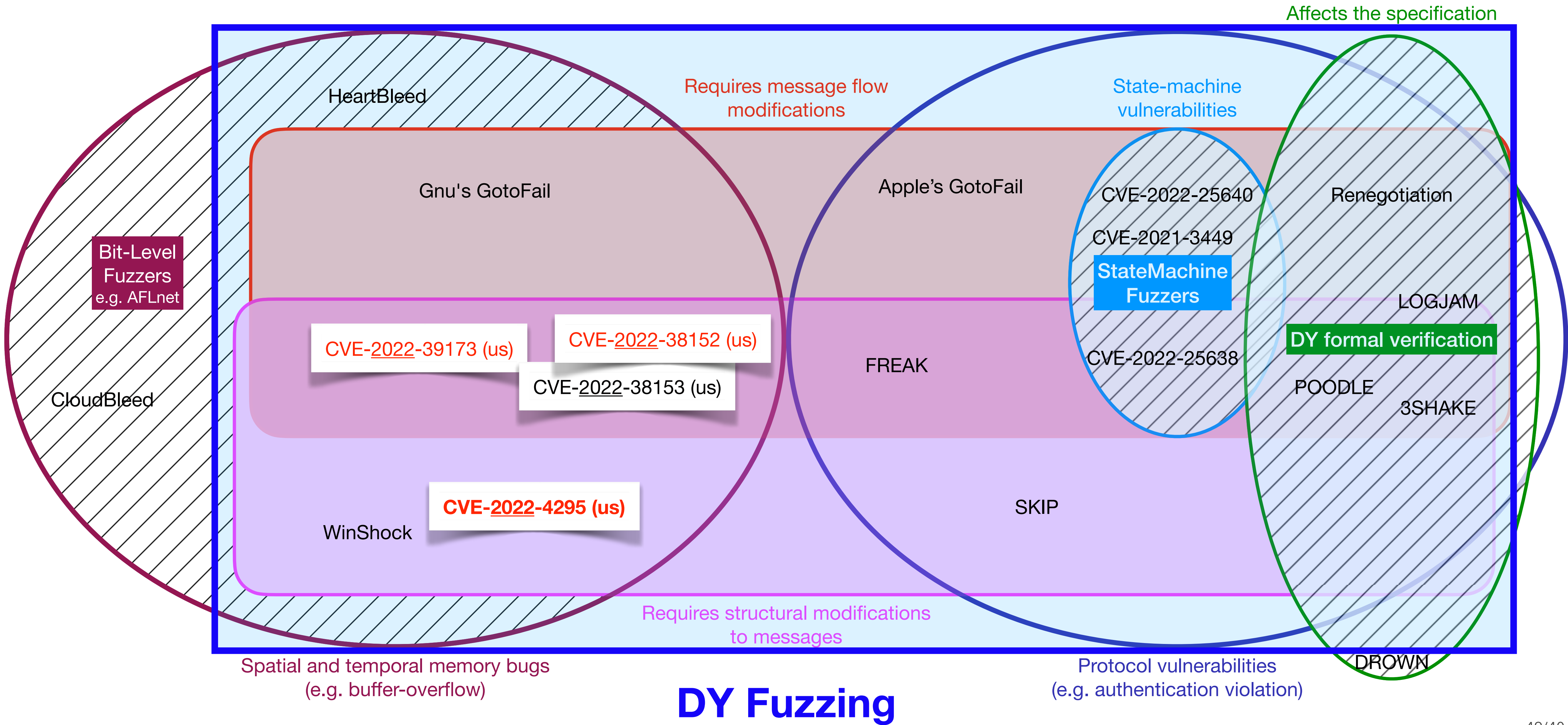
2019-2022



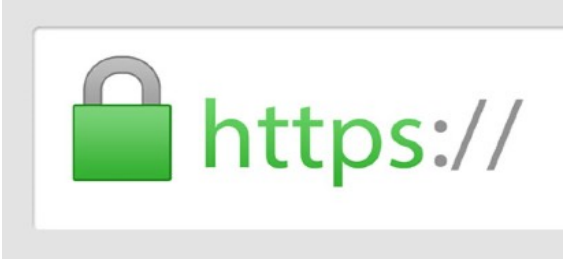
Retrospective of TLS Failures



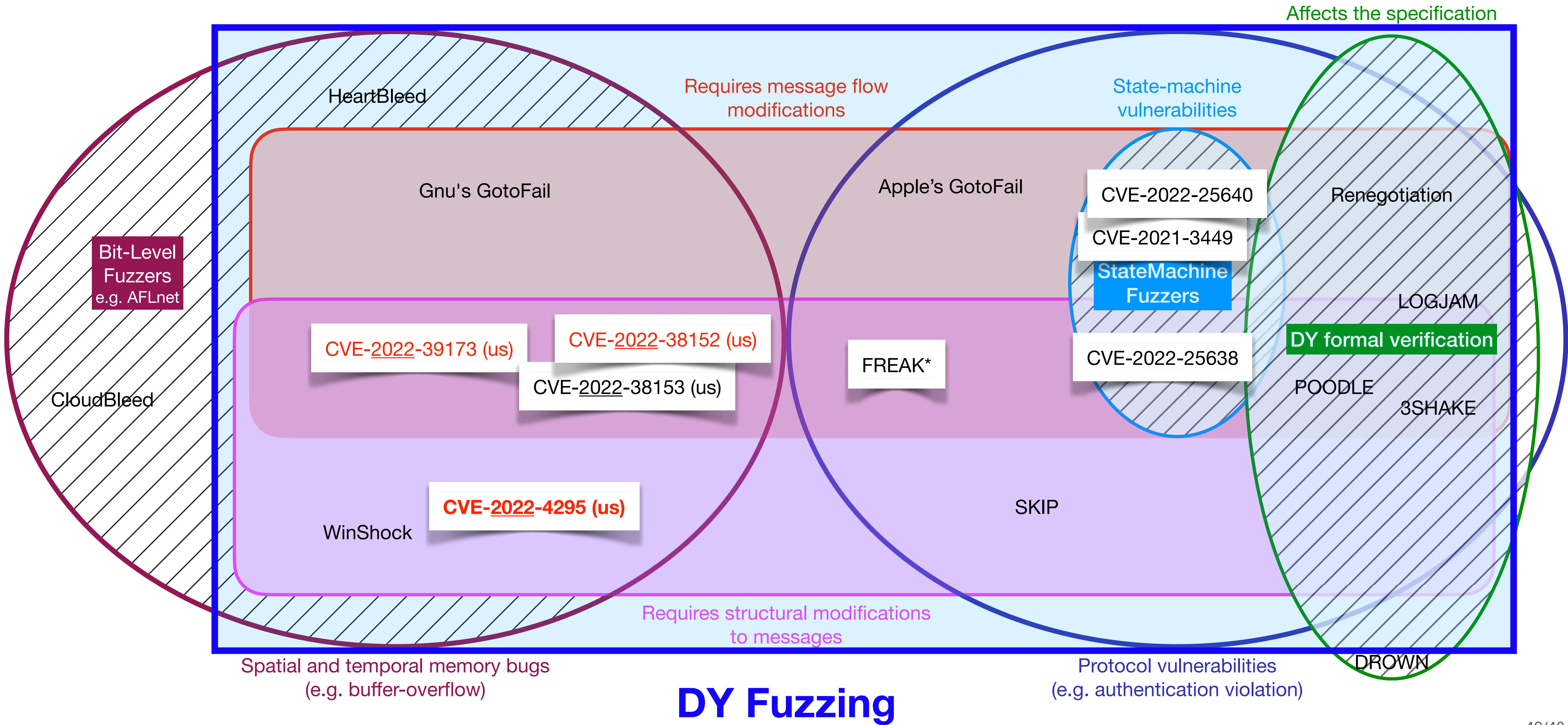
2019-2022



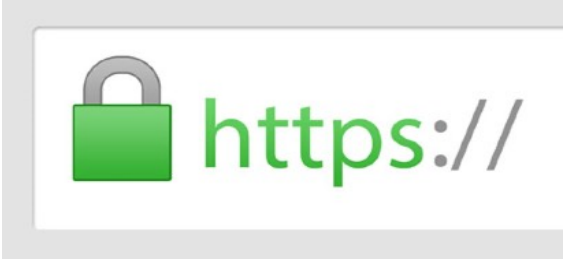
Retrospective of TLS Failures



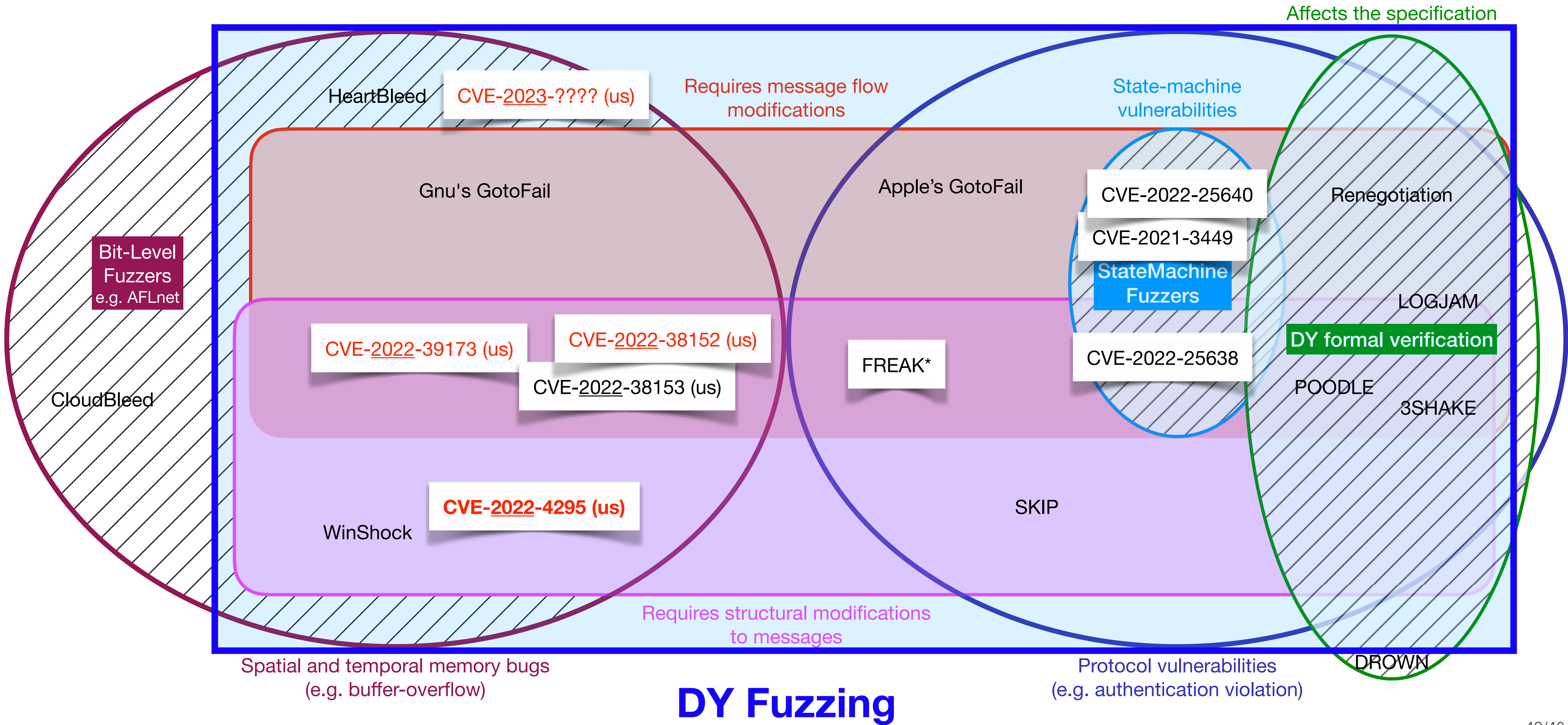
2019-2022



Retrospective of TLS Failures



2019-2022



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 $\text{WolfSSL}(t) \neq \text{OpenSSL}(t)$

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 $\text{WolfSSL}(t) \neq \text{OpenSSL}(t)$
- Or extend the oracle: more compromise scenarios, secrecy (abstraction, deduction?), privacy (approx.?), functional correctness / a model

Future Work (cont.)

Future Work (cont.)

- [WIP] 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.)

- **[WIP]** 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.)

- **[WIP]** 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)

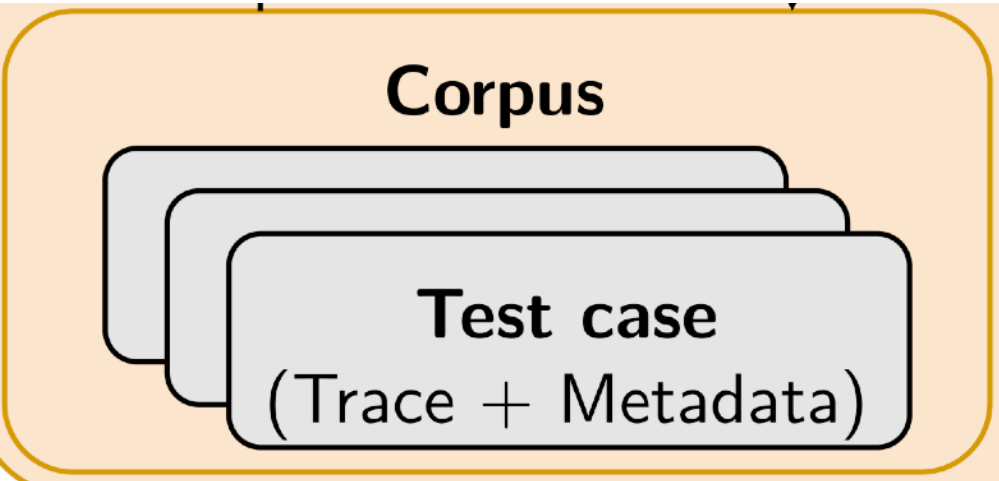
Future Work (cont.)

- **[WIP]** 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)

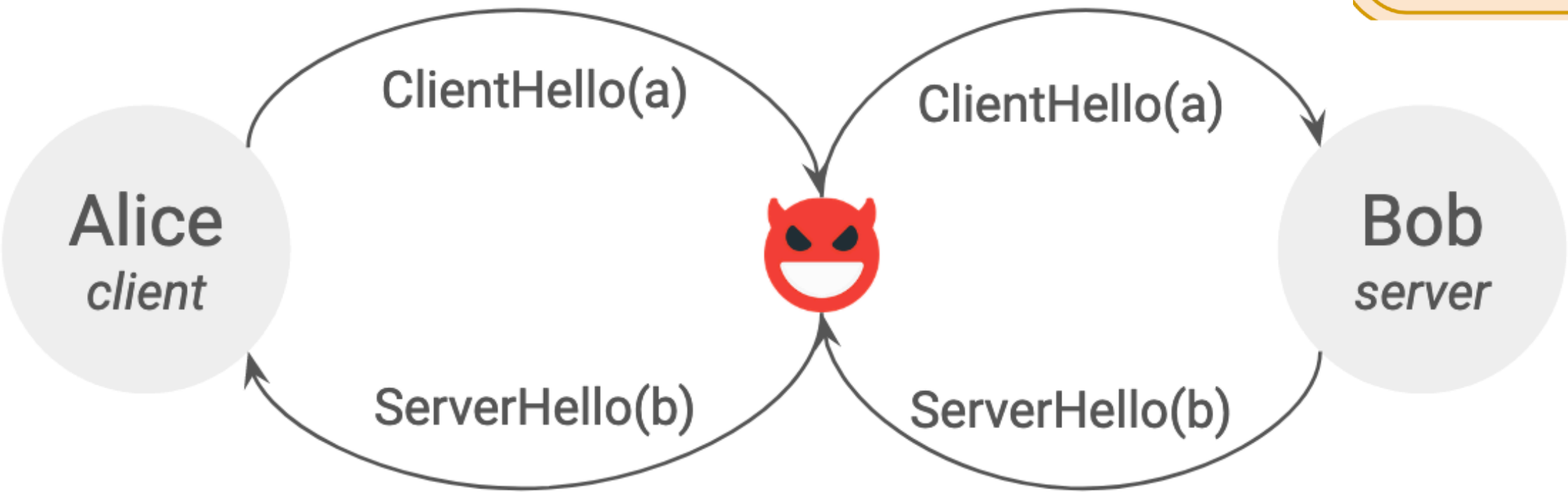
Long-Term

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

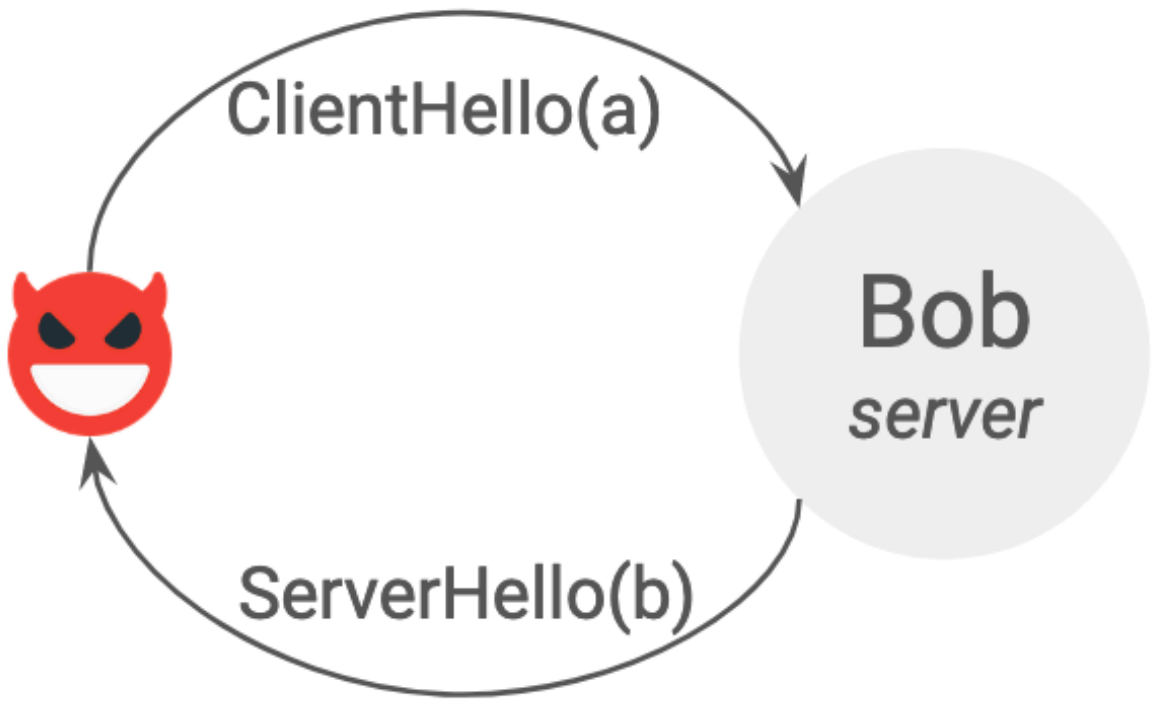
Seed Corpus



MITM Happy Flow

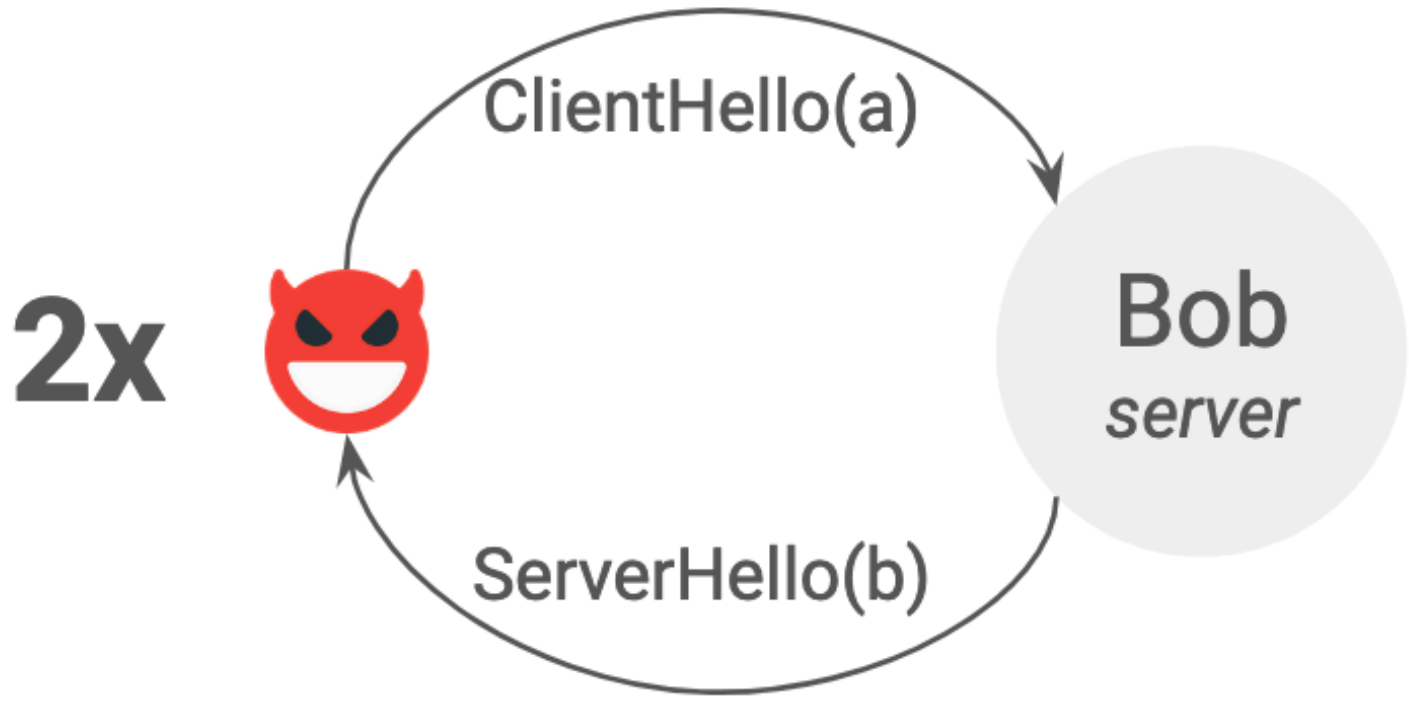


Client Attacker (Happy flow)



Session Resumption (Happy flow)

Similar to Client attacker, but performs a second handshake



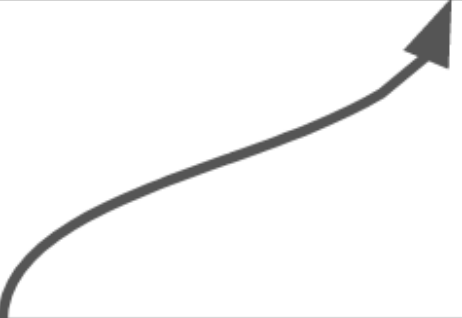
Tls puffin Terms Domain-Specific Language

```
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,  
        ...  
    )  
};
```

Tlsputtin Traces Domain-Specific Language

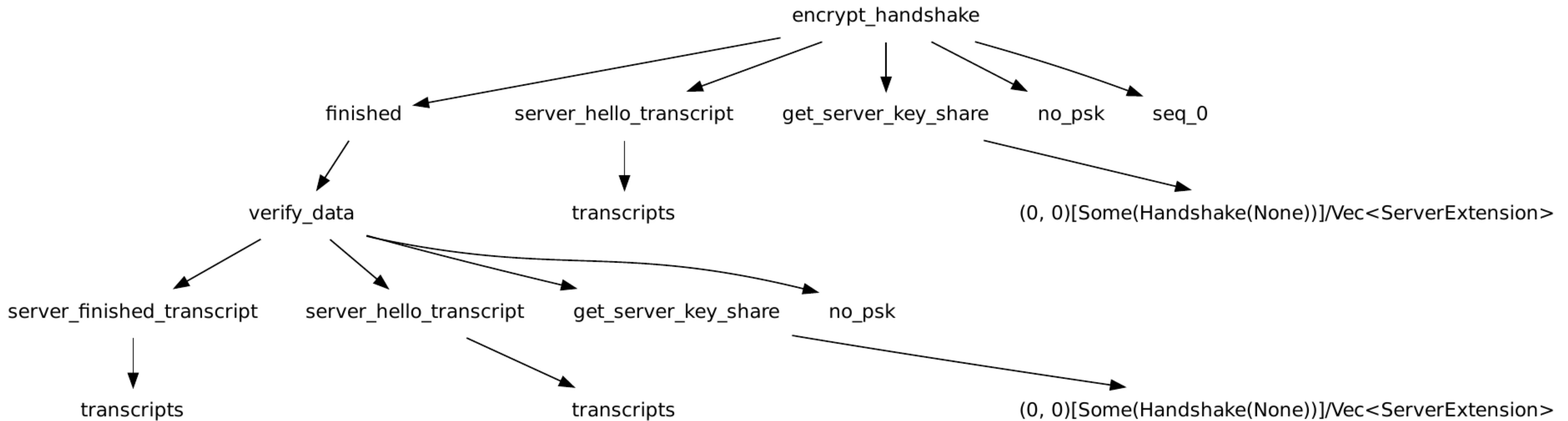
```
1  let recipe: Term = term! {  
2      fn_client_hello(  
3          fn_protocol_version12,  
4          fn_random,  
5          fn_session_id,  
6          fn_cipher_suites,  
7          fn_compressions,  
8          fn_client_extensions  
9      )  
10 };
```

Domain Specific Language
for declaring terms



```
1  let client_name = AgentName::new();  
2  let server_name = client::next();  
3  
4  let steps: Vec<Steps> = vec![  
5      OutputAction::new_step(client_name),  
6      InputAction::new_step(  
7          server_name, recipe  
8      )  
9  ];  
10  
11 let trace: Trace = Trace {  
12     prior_traces: vec![],  
13     descriptors: vec![  
14         AgentDescriptor::  
15             new_client(client_name, V1_3),  
16         AgentDescriptor::  
17             new_server(server_name, V1_3)  
18     ],  
19     steps  
20 };
```

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 $\langle m, \text{MAC}(h(\text{transcript}), k) \rangle$