Status Report

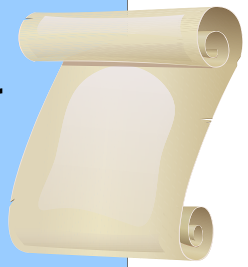
Formal Analysis of Web Security

Karthikeyan Bhargavan[1], Abhishek Bichhawat[2],
Quoc Huy Do[3], Daniel Fett[3], Ralf Küsters[3], Guido Schmitz[3]

1: INRIA, France
2: Carnegie Mellon University, USA
3: University of Stuttgart, Germany

Contents

Previous Work:
Generic formal pen-and-paper
model and proofs



- Comprehensive model in focus
- Not constrained by tools
- Not necessarily easy to use tools

Plan:
Mechanized
model and proofs

- Automation
- Executable model
- Testing

Previous Work

[SP 2014, ESORICS 2015, CCS 2015, CCS 2016, CSF 2017]

- Development of a generic and comprehensive formal model of the web infrastructure
  (more details later)
- Formal analysis of Mozilla's BrowserID
  Main design goal: privacy
  - Found severe attacks: Identity Injection Attack, PostMessage-Based Attack,
  - Proposed fixes for authentication and proved security
  - Privacy broken beyond repair
- Designed our own new SSO system: SPRESSO (https://spresso.me)
  Provably provides strong authentication and privacy properties.

Previous Work

[SP 2014, ESORICS 2015, CCS 2015, CCS 2016, CSF 2017]

- Analysis of OAuth 2.0
  - Found attacks: 307 Redirect Attack, IdP Mix-Up Attack, State Leak Attack, Naive RP Session Integrity Attack
  - Proposed fixes and proved security

Let's also discuss:
Current state of fixes
[draft-ietf-oauth-mix-up-mitigation-01]
[draft-ietf-oauth-security-topics-04]

- OpenID Connect 1.0 with Discovery and Dyn
  - Developed formal model of the standard
  - Proposed security guidelines mitigating known attacks
  - Proved security for (fixed) standard

All details: TR available at https://sec.uni-stuttgart.de

Formal Analysis of Web Applications and Standards

The web is complex ...

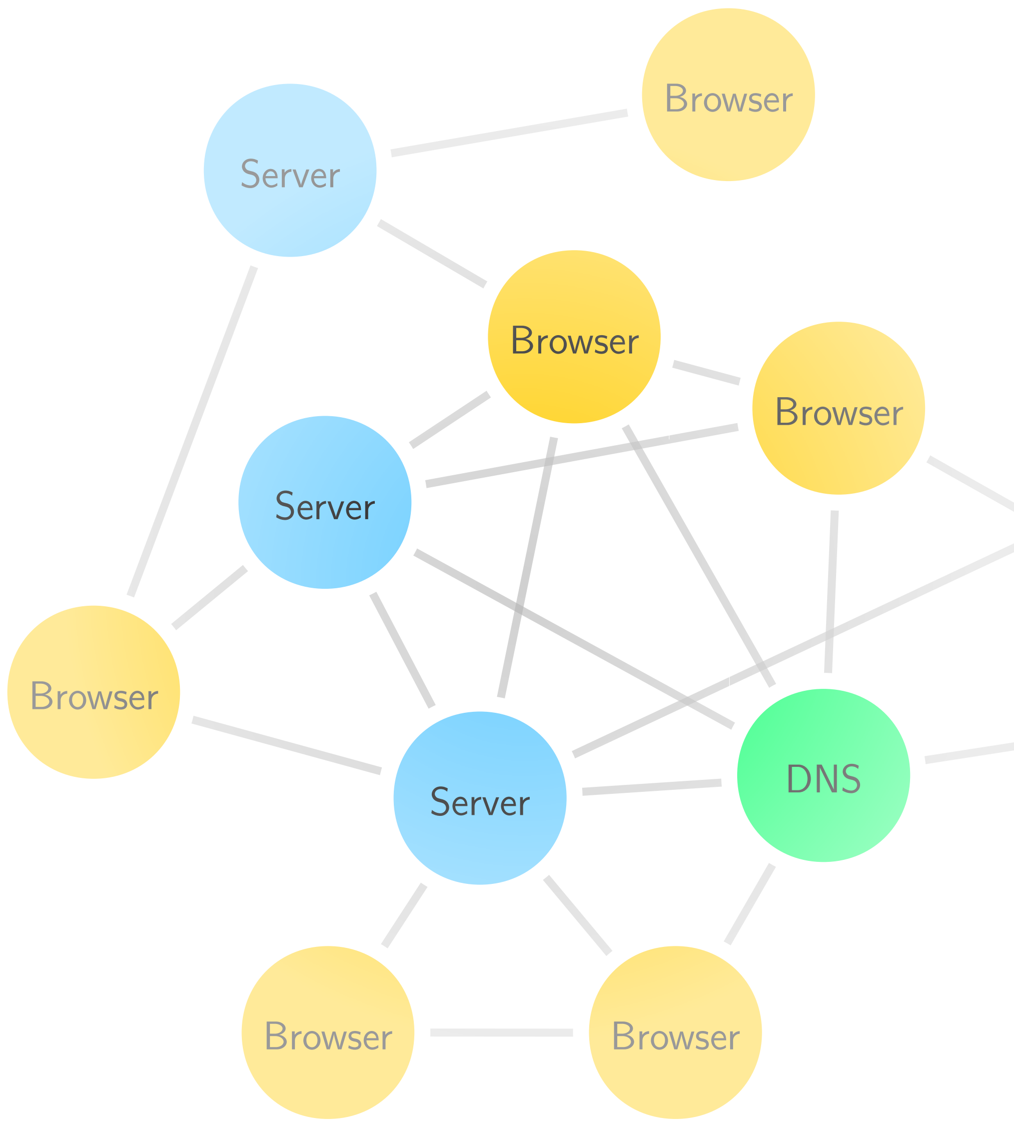
- Interaction of different components
- Large number of complex standards developed at a high pace by many separate organizations

... and web applications as well ...

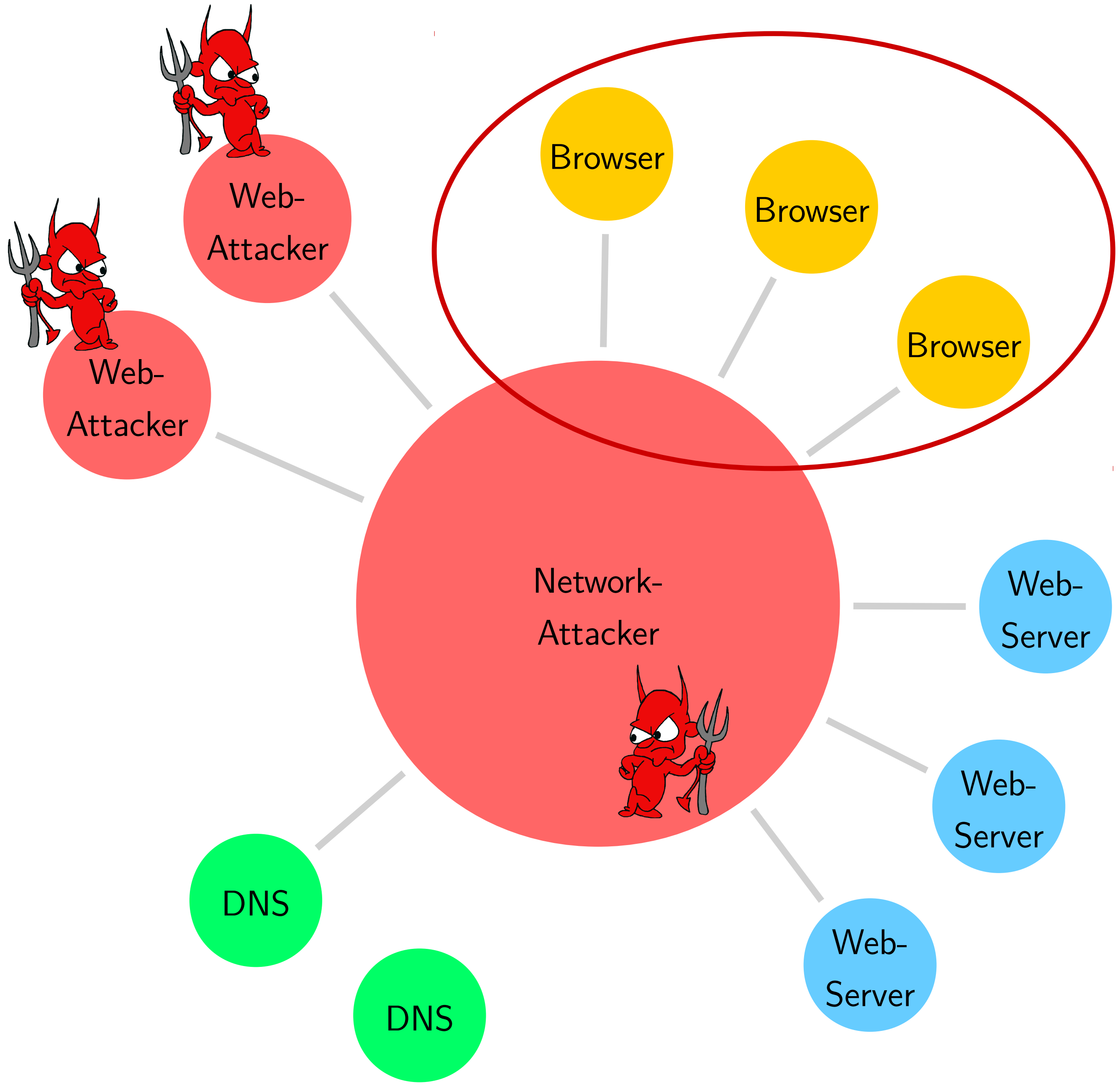
- Increasing complexity of web applications
- Many vulnerabilities

Formal methods enable us to ...

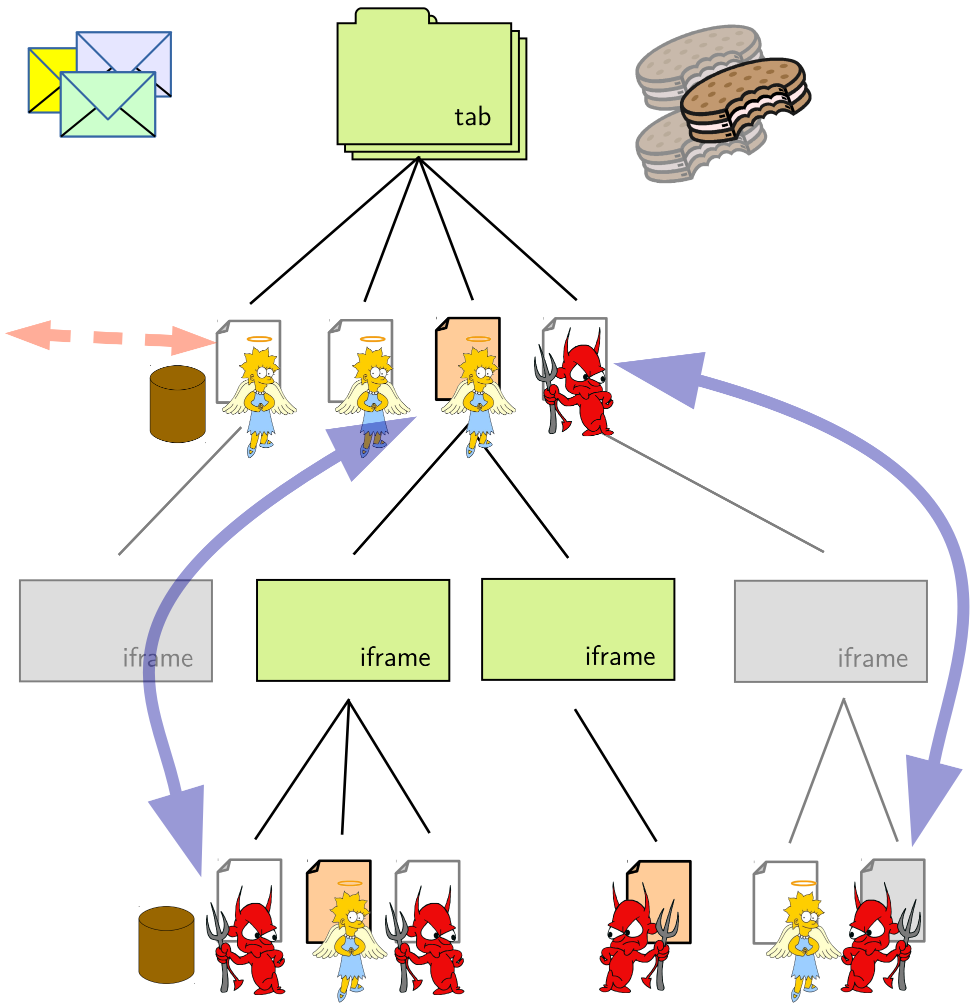
- develop a coherent model of core aspects of the web
- precisely specify security properties
- carry out security proofs



Network Model



Web Browser Model



Including ...

- DNS, HTTP, HTTPS
- window & document structure
- scripts
- attacker scripts
- web storage & cookies
- web messaging & XHR
- message headers
- redirections
- security policies
- dynamic corruption
- ...

Origin: https://example.com

Browser Model - Example

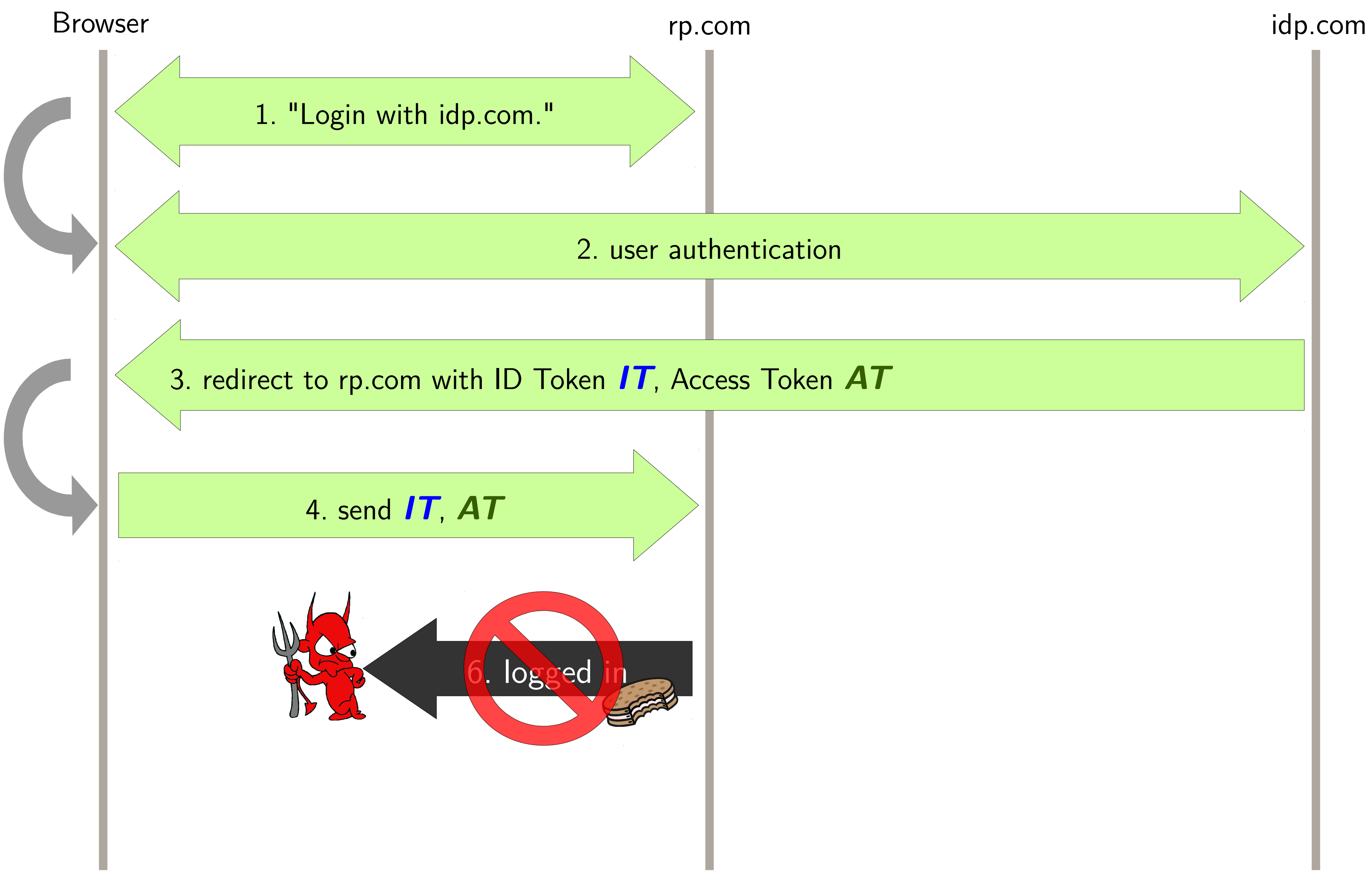
Algorithm 8 Web Browser Model: Process an HTTP response.

```
function PROCESSRESPONSE(response, reference, request, requestUrl, key, f, s')
    if Set-Cookie ∈ response.headers then
        for each c ∈⟨⟩ response.headers[Set-Cookie], c ∈ Cookies do
            let s'.cookies[request.host]
              ↪  := AddCookie(s'.cookies[request.host], c)
    if Strict-Transport-Security ∈ response.headers ∧ requestUrl.protocol ≡ S then
        let s'.sts := s'.sts +⟨⟩ request.host
    if Referer ∈ request.headers then
        let referrer := request.headers[Referer]
    else
        let referrer := ⊥
    if Location ∈ response.headers ∧ response.status ∈ {303, 307} then
        let url := response.headers[Location]
        if url.fragment ≡ ⊥ then
            let url.fragment := requestUrl.fragment
        let method' := request.method
        let body' := request.body
        if Origin ∈ request.headers then
            let origin := ⟨request.headers[Origin], ⟨request.host, url.protocol⟩⟩
        else
            let origin := ⊥
        if response.status ≡ 303 ∧ request.method ∉ {GET, HEAD} then
            let method' := GET
            let body' := ⟨⟩
```

Security Property – OIDC

- Authentication: a network attacker (and therefore also web attackers) should be unable to log in as an honest user at an honest RP using an honest IdP.
- Authorization: a network attacker should not be able to obtain or use a protected resource available to some honest RP at an IdP for some user unless certain parties involved in the authorization process are corrupted
- Session integrity: an attacker should be unable to forcefully log a user/browser in at some RP

Authentication Property of OIDC



Authentication Property of OIDC – Formal Definition

Let OIDC^n be an OIDC web system with a network attacker. We say that OIDC^n *is secure w.r.t. authentication* iff for every run ρ of OIDC^n, every configuration (S, E, N) in ρ, every $r \in \mathsf{RP}$ that is honest in S, every browser b that is honest in S, every identity $id \in \mathsf{ID}$ with $\mathsf{governor}(id)$ being an honest IdP, every service session identified by some nonce n for id at r, we have that n is not derivable from the attackers knowledge in S (i.e., $n \notin d_\emptyset(S(\mathsf{attacker}))$).

Limitations

- No language details
- No user interface details
- No byte-level attacks (e.g., buffer overflows)
- Abstract view on cryptography and TLS (Dolev-Yao Model)

Limitations

- Main limitation: pen-and-paper model and proof
  - Laborious
  - Error-prone
  - Non-executable

Contents

Previous Work:
Generic formal pen-and-paper model and proofs

- Comprehensive model in focus
- Not constrained by tools
- Not necessarily easy to use tools

Plan:
Mechanized model and proofs

- Automation
- Executable model
- Testing

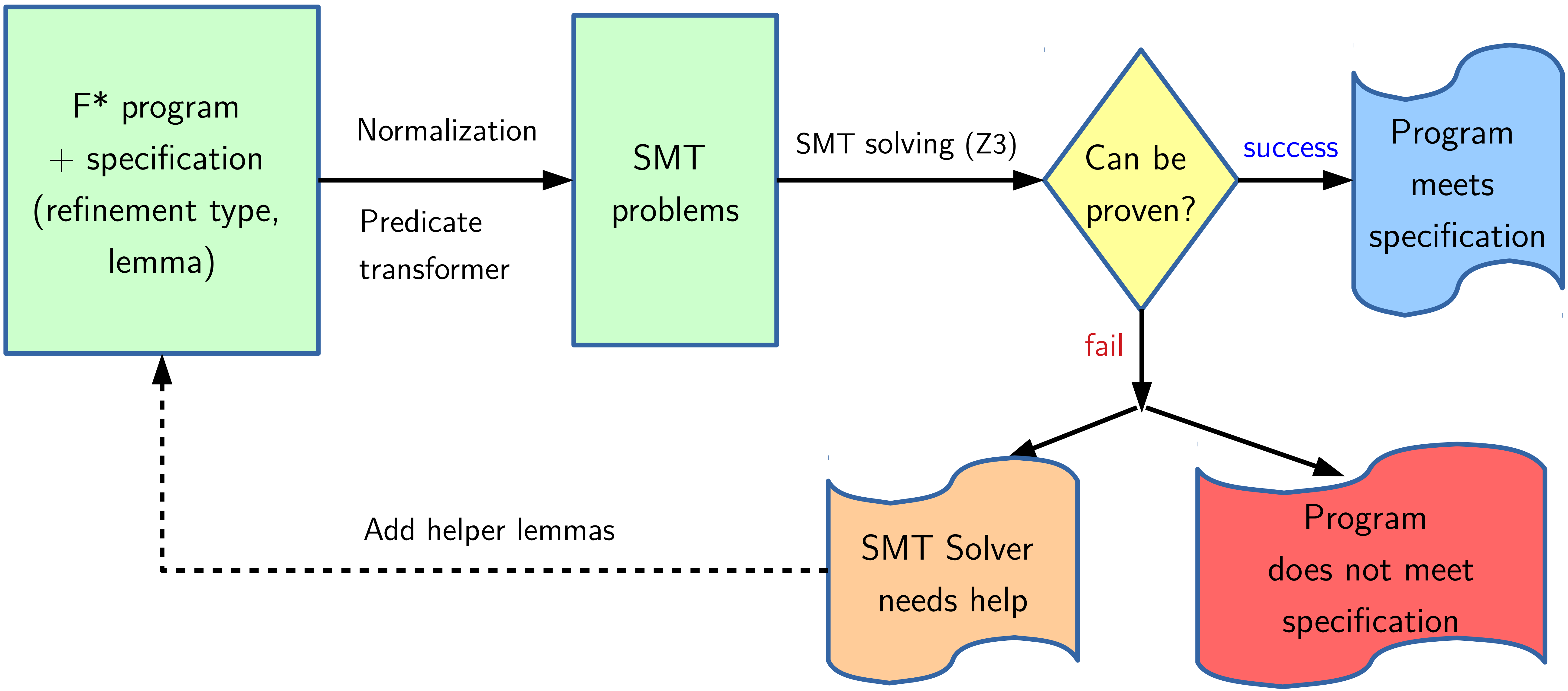
Mechanizing Model and Proofs: Approaches

- Fully automatic tools (ProVerif, Tamarin, Avispa)
  - Need more abstraction
  - Not adequate for a comprehensive model (complex data structure)
- Theorem prover-based approach
  - More precise
  - Can require user's interaction
  - More adequate for comprehensive model

What is F*?

- Functional programming language aimed at program verification
- Type system for specifying properties
- SMT Solver Z3 as the backend
- F* program can be translated to OCaml, F#, C, or JS

How F* Works



F* program
+ specification
(refinement type,
lemma)
Normalization
Predicate
transformer
SMT
problems
SMT solving (Z3)
Can be
proven?
success
Program
meets
specification
fail
SMT Solver
needs help
Program
does not meet
specification
Add helper lemmas

Why F*?

- Seems adequate for encoding our comprehensive model
  - Pure functional programming language
  - Sufficient for modeling complex data structures (browsers, servers)
  - Rich, versatile type system expressing precise, compact security properties
  - Powerful type checker enables some automation
  - Translation into executable code (also for sanity check, testing)
- Actively supported
- Strength proven in practice (TLS)

F* - Simple Example

```
val factorial: int -> int
let rec factorial n =
  if n<=1 then 1 else n * (factorial (n-1))
```

F* - Simple Example

```
val factorial: n:int{n>=0} -> x:int{x>=0}
let rec factorial n =
    if n<=1 then 1 else n * (factorial (n-1))


val factorial_lemma: n:int{n>2} -> Lemma ( factorial n >n)
let  factorial_lemma n = ()
```

F* - Simple Example

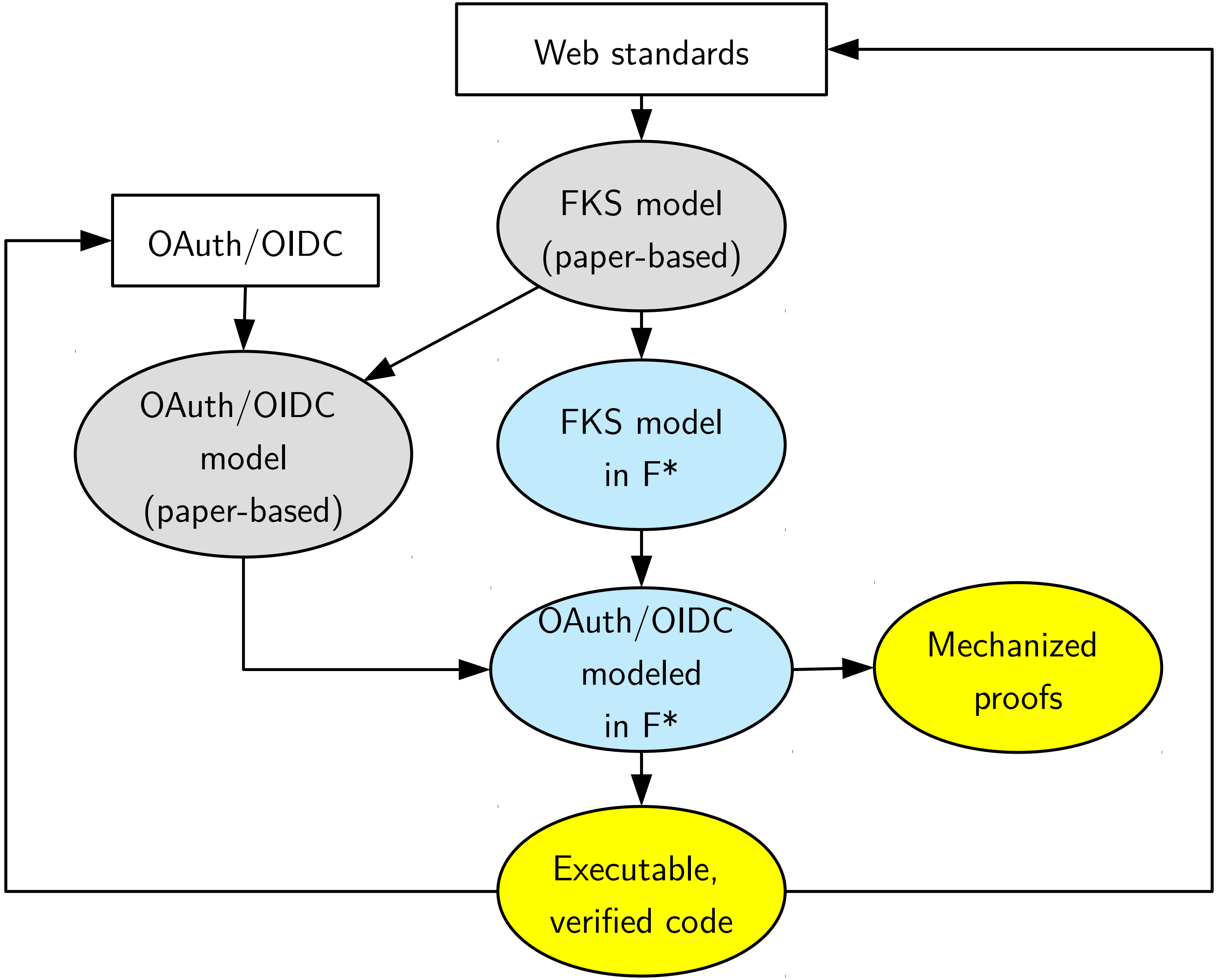
```
val factorial: n:int{n>=0} -> x:int{x>=0}
let rec factorial n =
    if n<=1 then 1 else n * (factorial (n-1))


val factorial_lemma: n:int{n>2} -> Lemma ( factorial n >n)
let rec  factorial_lemma n = match n with
    |3 -> ()
    |_ -> factorial_lemma (n-1)
```

F* - Simple Example

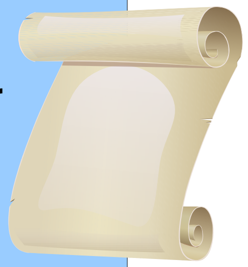
- Demo

Working Plan



Conclusion

Previous Work:
Generic formal pen-and-paper
model and proofs



- Comprehensive model in focus
- Not constrained by tools
- Not necessarily easy to use tools

Thank you!

Plan:
Mechanized
model and proofs

- Automation
- Executable model
- Testing