Refinement-Based Verification of Security Protocol Implementations

Linard Arquint, ETH Zurich & AR in Cloud Ops

Based on joint work with Felix A. Wolf, Joseph Lallemand, Ralf Sasse, Christoph Sprenger, Sven N. Wiesner, David Basin, Peter Müller, Samarth Kishor, Jason R. Koenig, Joey Dodds, and Daniel Kroening

Motivation

Protocol Model
in Tamarin

- Formal protocol description
- Automatic tools: Tamarin, ProVerif, …
- Successfully applied to many protocols

Motivation

Implementation

But:

- Implementation is executed, not model
- Is the protocol implemented correctly?
- Does it contain vulnerabilities?
- Are there any runtime errors?

Our Approach

Verification of
Protocol Models
in Tamarin

Verification of
Implementations
in Program Verifiers

Our Approach

Verification of
Protocol Models
in Tamarin

Verification of
Implementations
in Program Verifiers

Property Preservation

Protocol Model
in Tamarin

1

satisfies

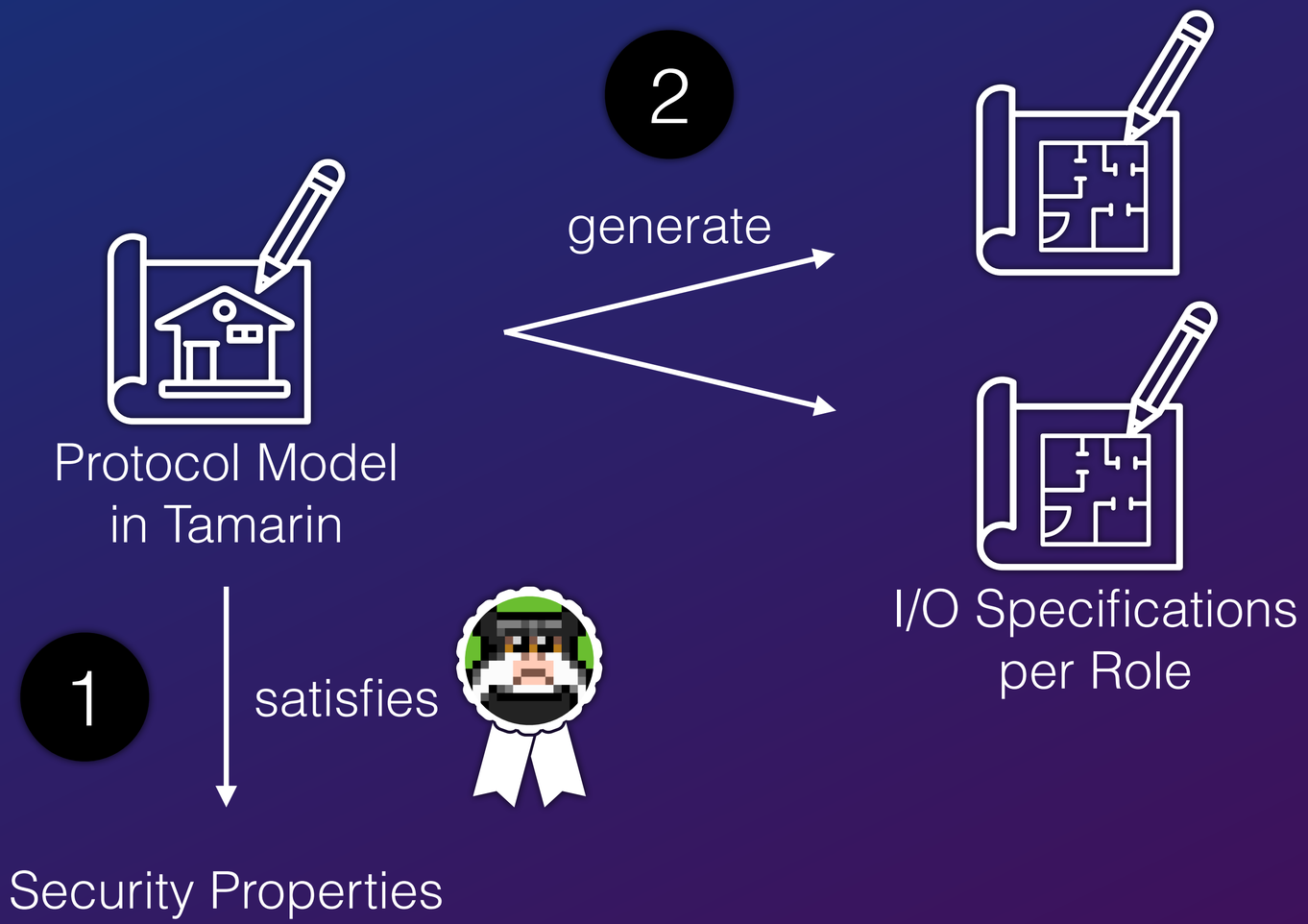
Security Properties

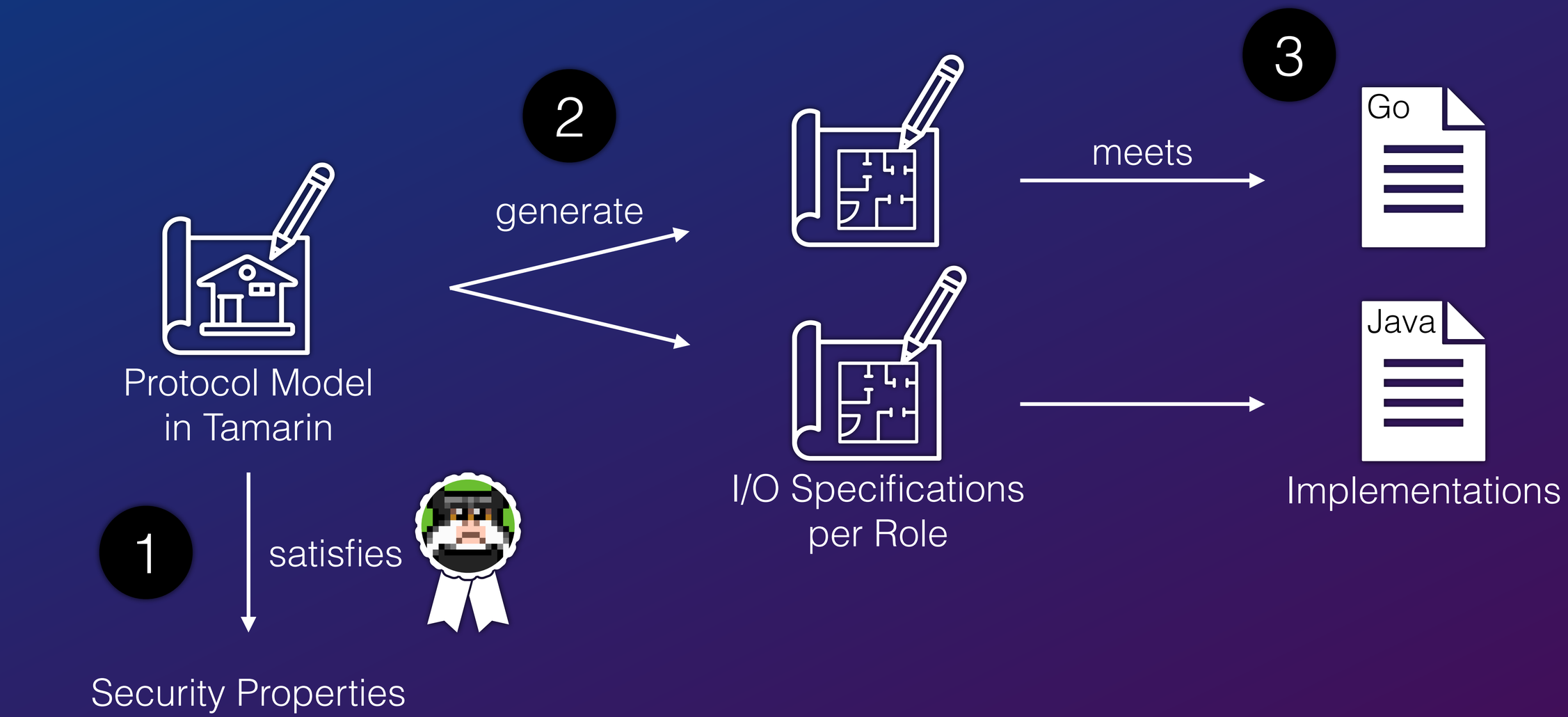
Protocol Model
in Tamarin

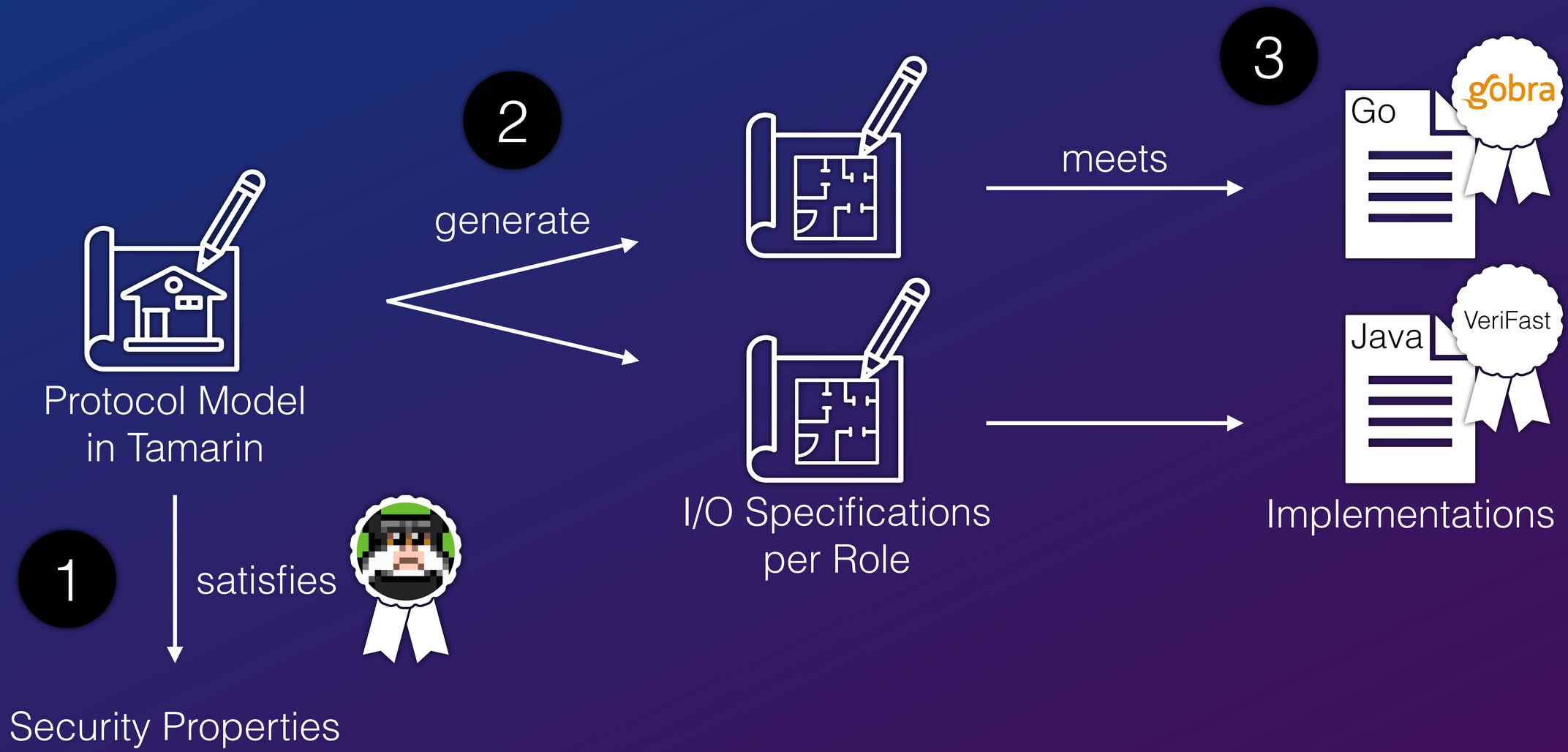
1

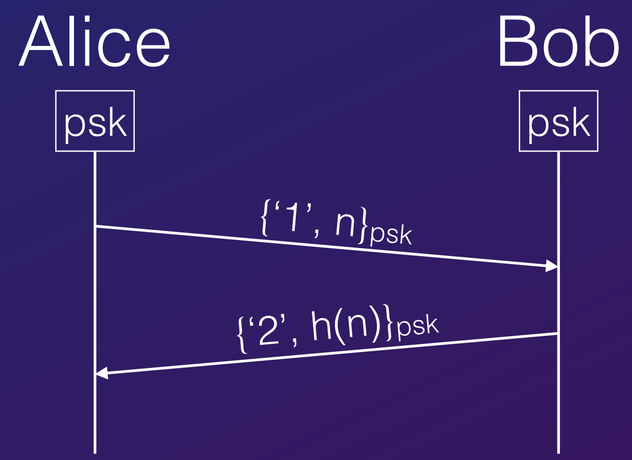
satisfies

Security Properties

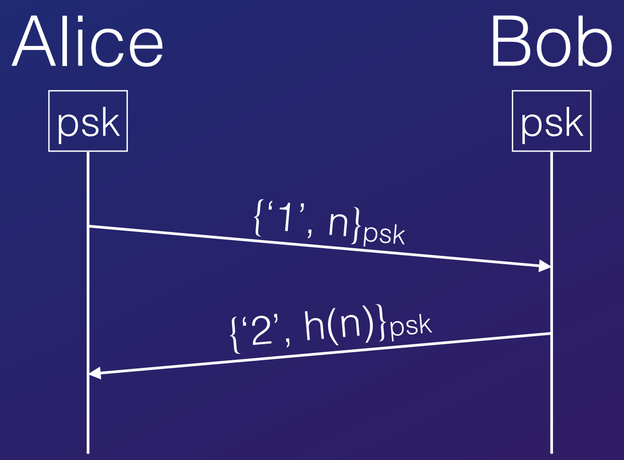

2
generate
Protocol Model
in Tamarin
I/O Specifications
per Role
1
satisfies
Security Properties

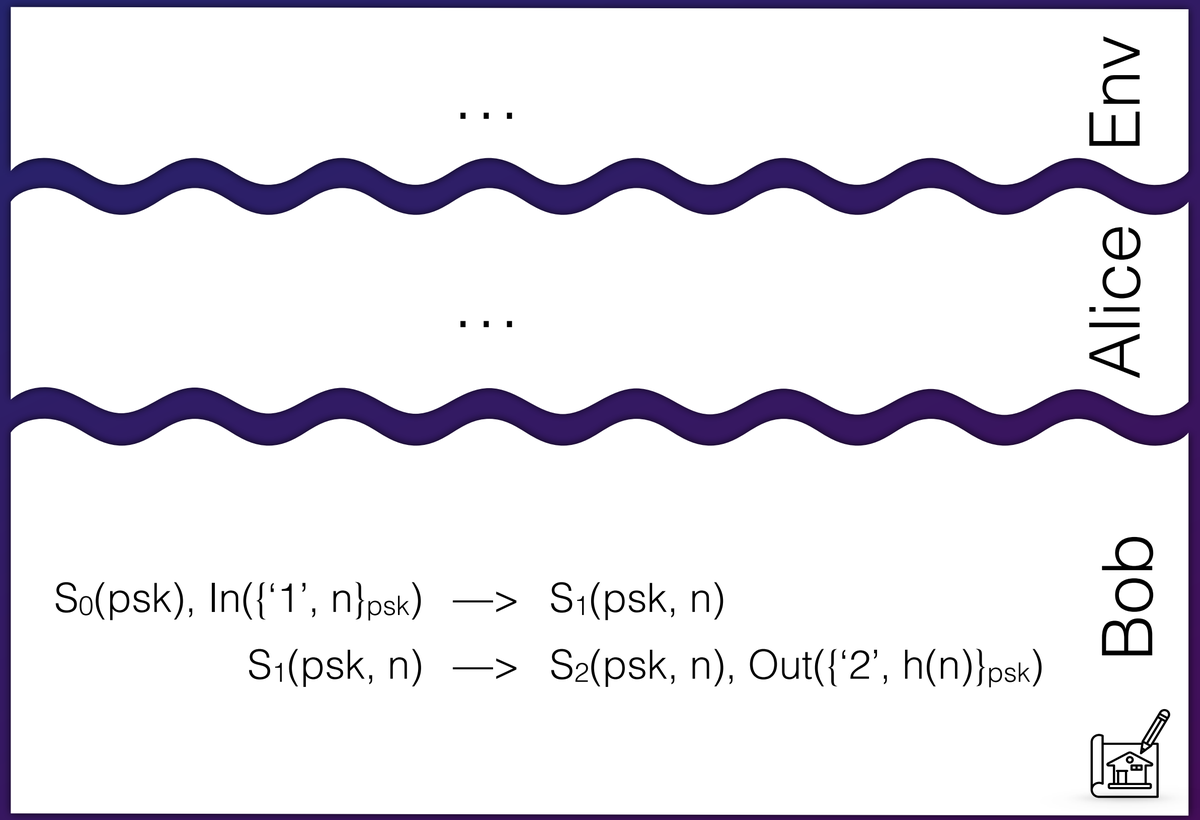

3
2
meets
Go
generate
Java
Protocol Model
in Tamarin
I/O Specifications
per Role
Implementations
1
satisfies
Security Properties


3
2
generate
meets
Go
gobra
Protocol Model
in Tamarin
VeriFast
Java
1
satisfies
I/O Specifications
per Role
Implementations
Security Properties


Alice
Bob
psk
psk
{‘1’, n}psk
{‘2’, h(n)}psk

1


Alice
Bob
psk
psk
{‘1’, n}psk
{‘2’, h(n)}psk


Env
Alice
Bob
...
...
S0(psk), In({‘1’, n}psk) —> S1(psk, n)
S1(psk, n) —> S2(psk, n), Out({‘2’, h(n)}psk)

3

S_0(psk), In({‘1’, n}$_{psk}$) —> S_1(psk, n)

S_1(psk, n) —> S_2(psk, n), Out({‘2’, h(n)}$_{psk}$)

Bob

```
func main(psk []byte) {


  m1 := recv()

  // parse m1
  p1, ok := sdecrypt(m1, psk)
  if !ok { return }

  tag1, n := destruct(p1)
  if tag1 != 1 { return }









  p2 := construct(2, hash(n))
  m2 := sencrypt(p2, psk)



  send(m2)
}
```

3

$S_0(psk)$, $In(\{‘1’, n\}_{psk})$ —> $S_1(psk, n)$

$S_1(psk, n)$ —> $S_2(psk, n)$, $Out(\{‘2’, h(n)\}_{psk})$

Bob

```
requires Bob_IO_Spec()
func main(psk []byte) {


  m1 := recv()

  // parse m1
  p1, ok := sdecrypt(m1, psk)
  if !ok { return }

  tag1, n := destruct(p1)
  if tag1 != 1 { return }









  p2 := construct(2, hash(n))
  m2 := sencrypt(p2, psk)



  send(m2)
}
```

3

$S_0(psk)$, $In(\{‘1’, n\}_{psk})$ —> $S_1(psk, n)$

$S_1(psk, n)$ —> $S_2(psk, n)$, $Out(\{‘2’, h(n)\}_{psk})$

Bob

```
requires Bob_IO_Spec()
func main(psk []byte) {
  // obtain permission to receive
  unfold Bob_IO_Spec()
  m1 := recv()

  // parse m1
  p1, ok := sdecrypt(m1, psk)
  if !ok { return }

  tag1, n := destruct(p1)
  if tag1 != 1 { return }









  p2 := construct(2, hash(n))
  m2 := sencrypt(p2, psk)



  send(m2)
}
```

3

$S_0(psk)$, $In(\{‘1’, n\}_{psk})$ —> $S_1(psk, n)$

$S_1(psk, n)$ —> $S_2(psk, n)$, $Out(\{‘2’, h(n)\}_{psk})$

Bob

```
requires Bob_IO_Spec()
func main(psk []byte) {
  // obtain permission to receive
  unfold Bob_IO_Spec()
  m1 := recv()

  // parse m1
  p1, ok := sdecrypt(m1, psk)
  if !ok { return }

  tag1, n := destruct(p1)
  if tag1 != 1 { return }

  unfold Bob_IO_Spec()
  internal_op_1()



  p2 := construct(2, hash(n))
  m2 := sencrypt(p2, psk)



  send(m2)
}
```

3

$S_0(psk)$, $In(\{‘1’, n\}_{psk})$ —> $S_1(psk, n)$

$S_1(psk, n)$ —> $S_2(psk, n)$, $Out(\{‘2’, h(n)\}_{psk})$

Bob

```
requires Bob_IO_Spec()
func main(psk []byte) {
  // obtain permission to receive
  unfold Bob_IO_Spec()
  m1 := recv()

  // parse m1
  p1, ok := sdecrypt(m1, psk)
  if !ok { return }

  tag1, n := destruct(p1)
  if tag1 != 1 { return }

  unfold Bob_IO_Spec()
  internal_op_1()
  unfold Bob_IO_Spec()
  internal_op_2()

  p2 := construct(2, hash(n))
  m2 := sencrypt(p2, psk)



  send(m2)
}
```

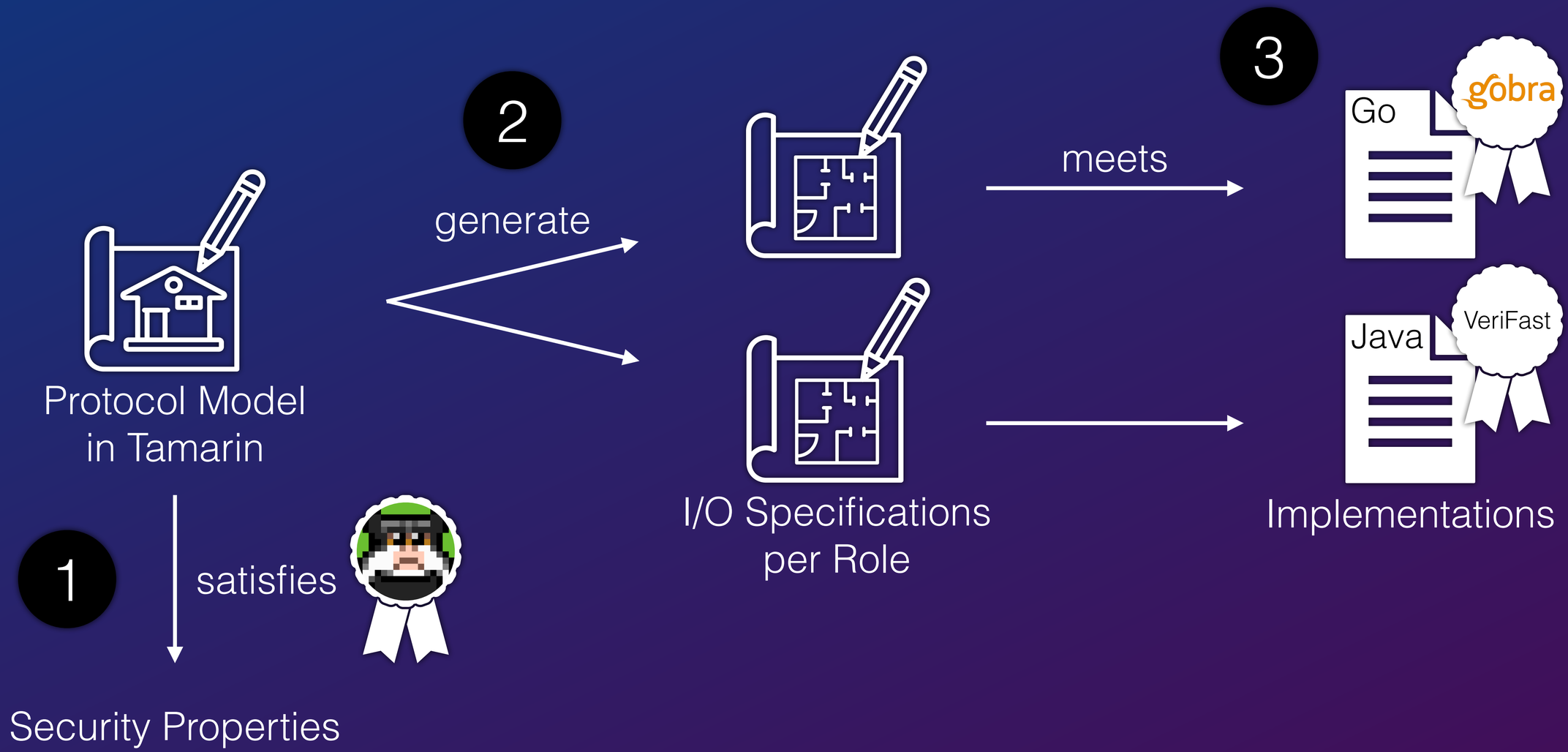
3

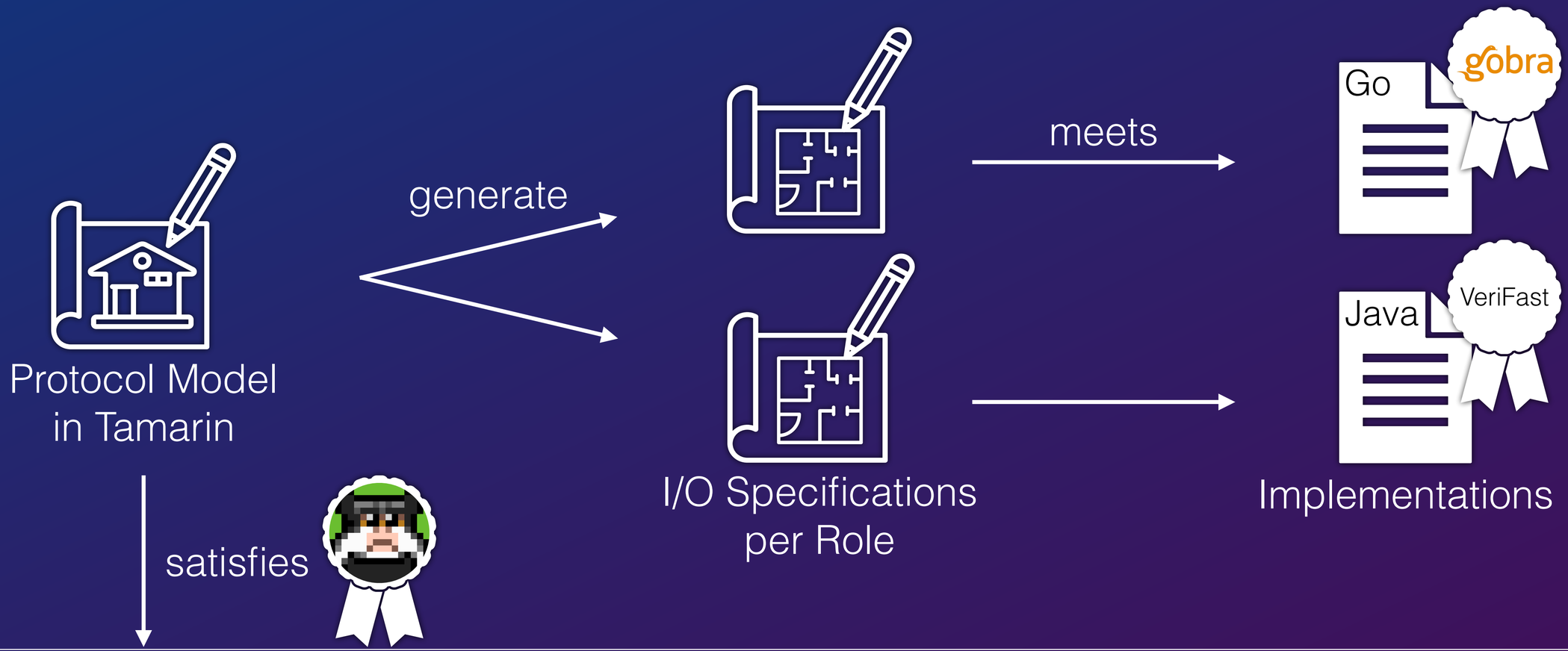
$S_0(psk)$, $In(\{‘1’, n\}_{psk})$ —> $S_1(psk, n)$

$S_1(psk, n)$ —> $S_2(psk, n)$, $Out(\{‘2’, h(n)\}_{psk})$

Bob

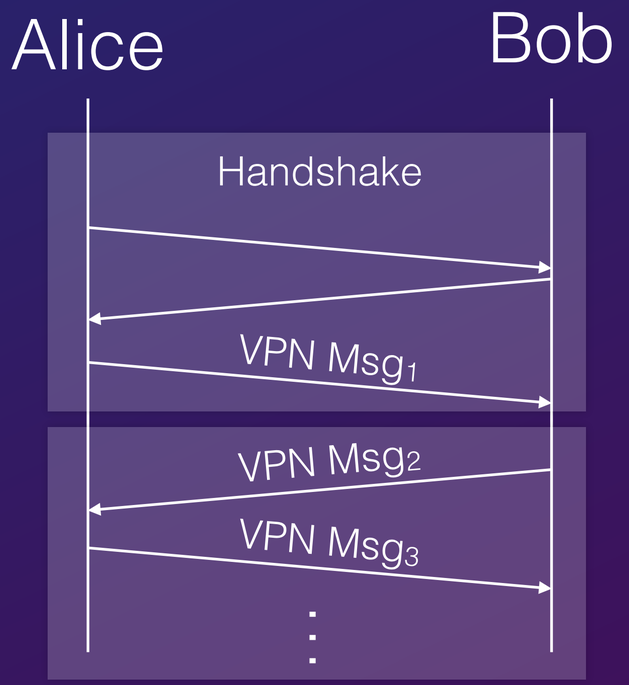
```
requires Bob_IO_Spec()
func main(psk []byte) {
  // obtain permission to receive
  unfold Bob_IO_Spec()
  m1 := recv()

  // parse m1
  p1, ok := sdecrypt(m1, psk)
  if !ok { return }

  tag1, n := destruct(p1)
  if tag1 != 1 { return }

  unfold Bob_IO_Spec()
  internal_op_1()
  unfold Bob_IO_Spec()
  internal_op_2()

  p2 := construct(2, hash(n))
  m2 := sencrypt(p2, psk)

  // obtain permission to send
  unfold Bob_IO_Spec()
  send(m2)
}
```


3
2
generate
meets
Go
gobra
Protocol Model
in Tamarin
VeriFast
Java
1
satisfies
I/O Specifications
per Role
Implementations
Security Properties


Protocol Model
in Tamarin
generate
I/O Specifications
per Role
meets
Go
gobra
Java
VeriFast
Implementations
satisfies
Security Properties
⊇
Tr(Specifications)
⊇
Tr(Implementations)

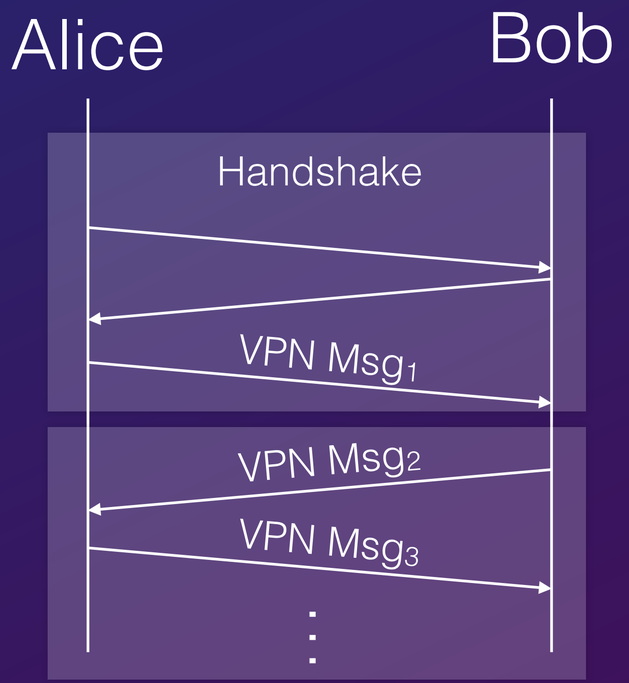
WIREGUARD®
FAST, MODERN, SECURE VPN TUNNEL

- VPN protocol consisting of handshake & transport phase



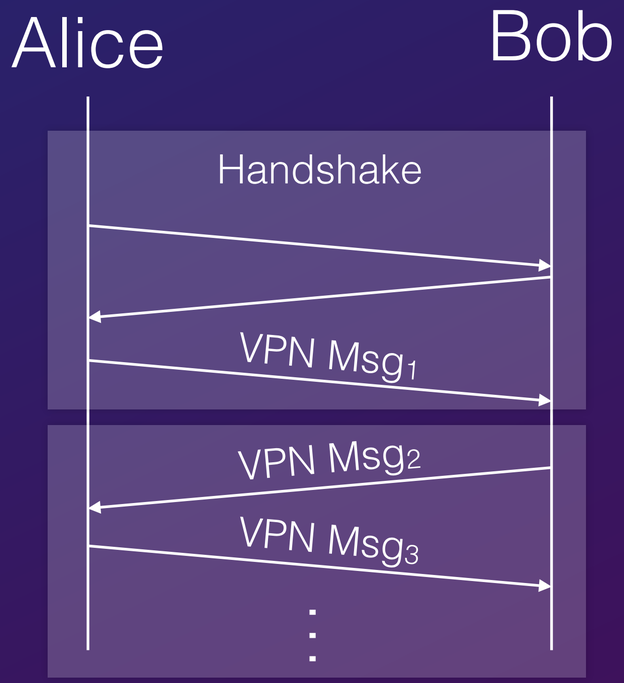
WIREGUARD®
FAST, MODERN, SECURE VPN TUNNEL

- VPN protocol consisting of handshake & transport phase
- ~400 LoC Tamarin model



WIREGUARD®
FAST, MODERN, SECURE VPN TUNNEL

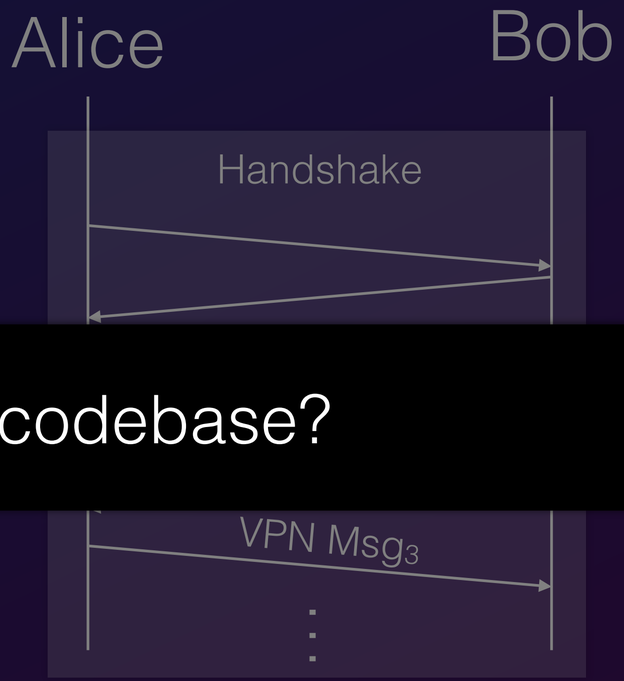
- VPN protocol consisting of handshake & transport phase
- ~400 LoC Tamarin model
- ~600 LoC Go code
- ~1.2k lines of generated I/O spec
- ~2.8k lines of proof annotations



gobra

WIREGUARD®
FAST, MODERN, SECURE VPN TUNNEL

- VPN protocol consisting of handshake & transport phase



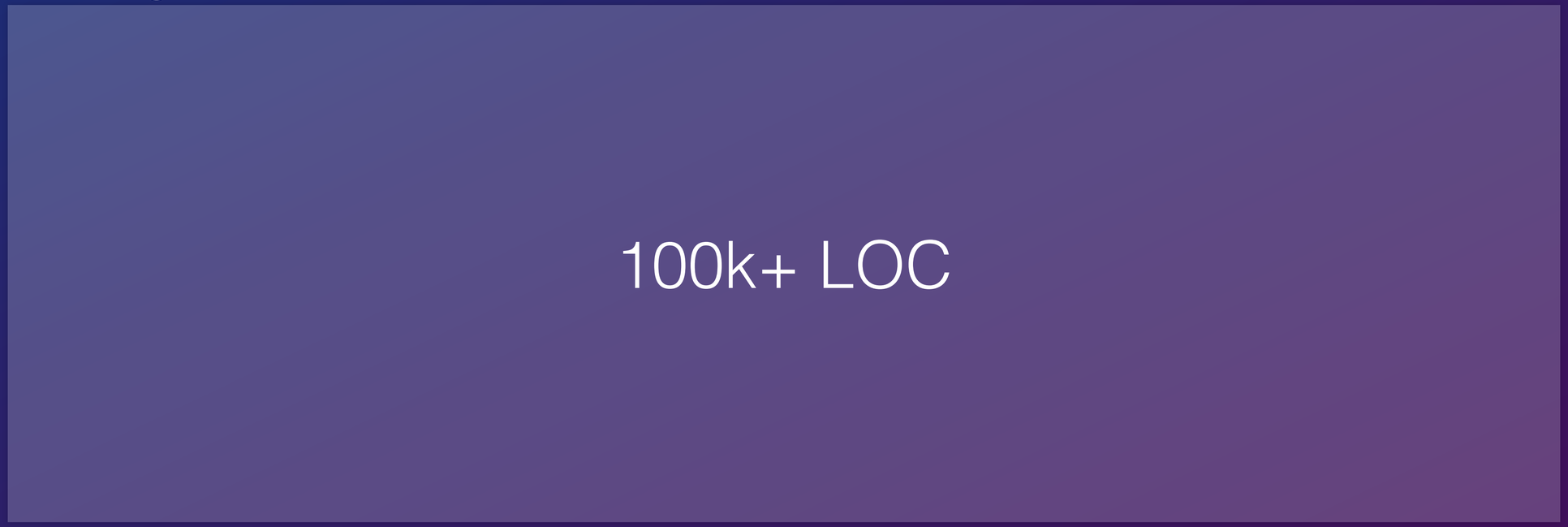
Do we have to verify the entire codebase?

- ~600 LoC Go code
- ~1.2k lines of generated I/O spec
- ~2.8k lines of proof annotations

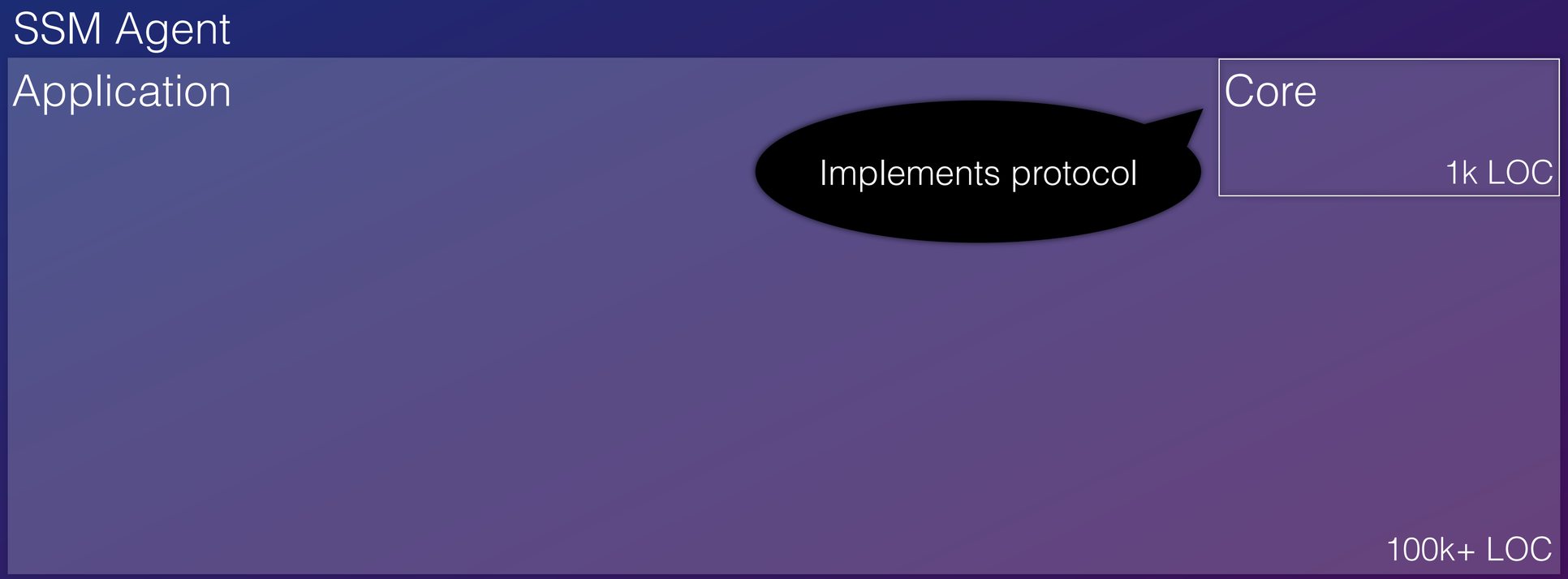
gobra

Extensions

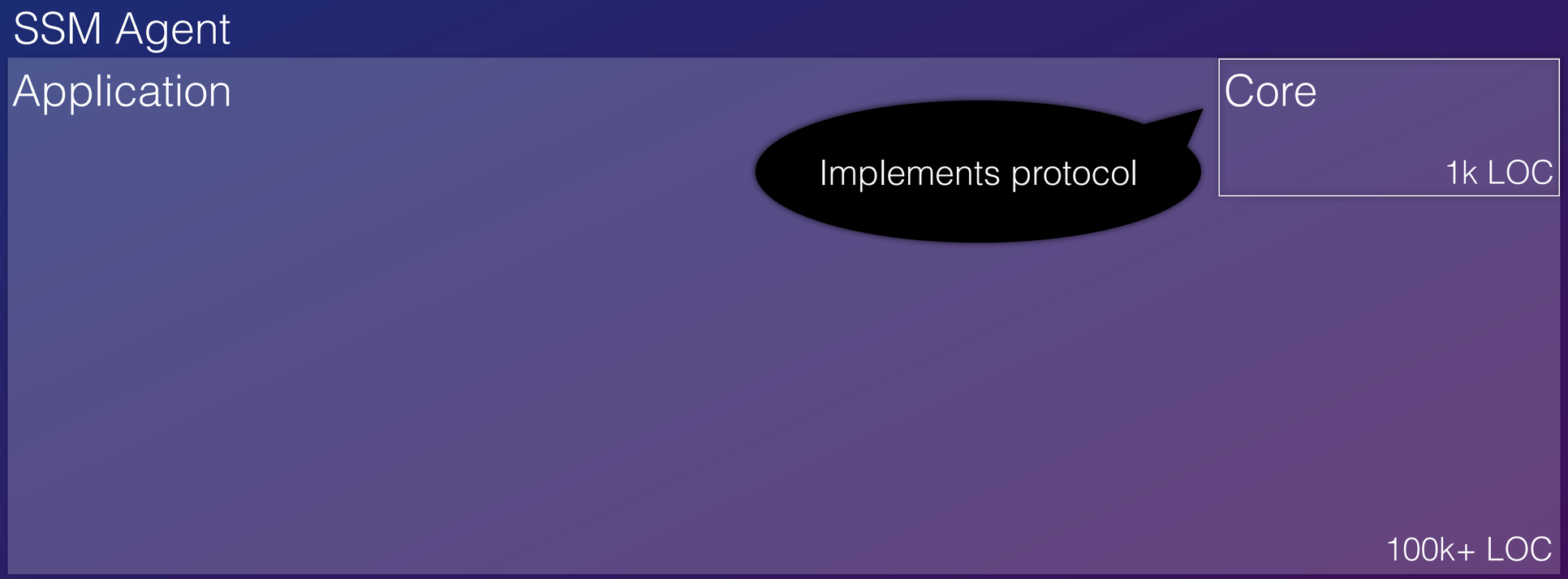
SSM Agent



Extensions


SSM Agent
Application
Core
1k LOC
Implements protocol
100k+ LOC

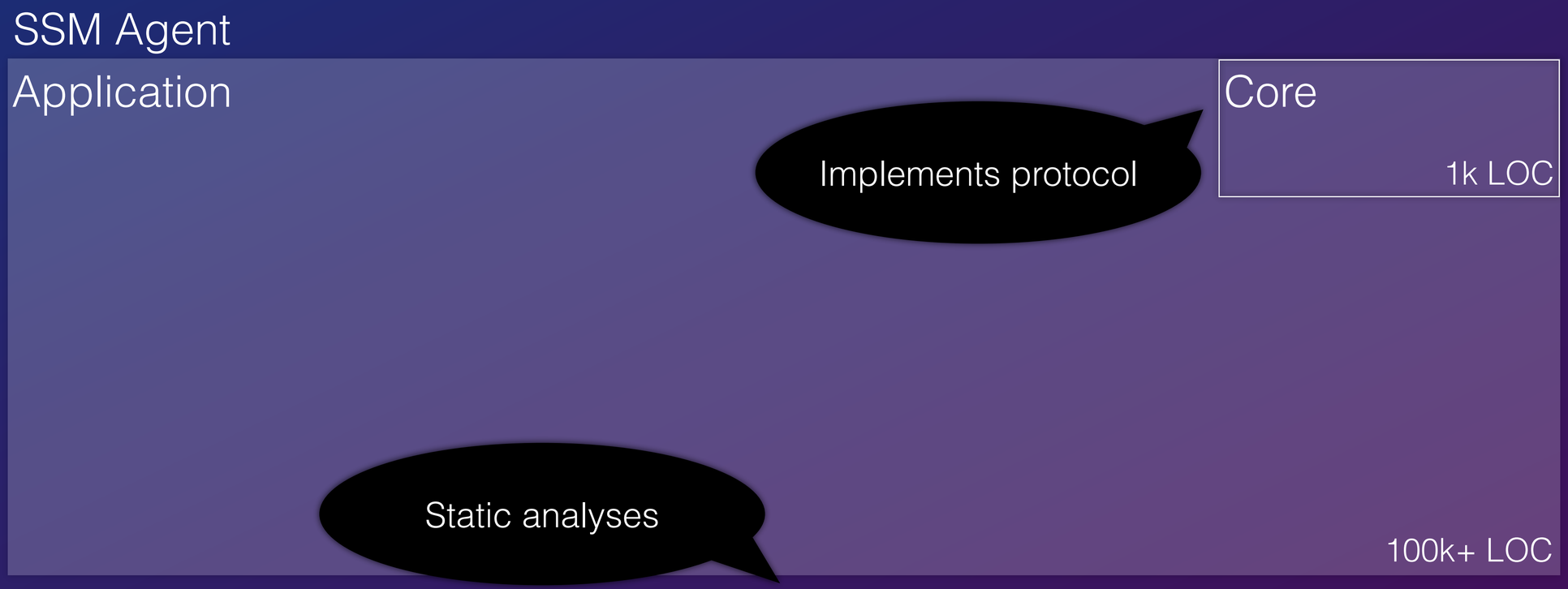
Extensions



What I/O operations can we safely allow in Application?

How do we ensure Application respects Core's preconditions?

Extensions



What I/O operations can we safely allow in Application?

How do we ensure Application respects Core's preconditions?

Extensions

SSM Agent

Application

Implements protocol

Core

1k LOC

100k+ LOC

What I/O

n?

How do v

nditions?

The Secrets Must Not Flow: Scaling Security Verification to Large Codebases

Linard Arquint
Department of Computer Science
ETH Zurich, Switzerland

Samarth Kishor
Amazon Web Services, USA

Jason R. Koenig
Amazon Web Services, USA

Joey Dodds
Amazon Web Services, USA

Daniel Kroening
Amazon Web Services, USA

Peter Müller
Department of Computer Science
ETH Zurich, Switzerland

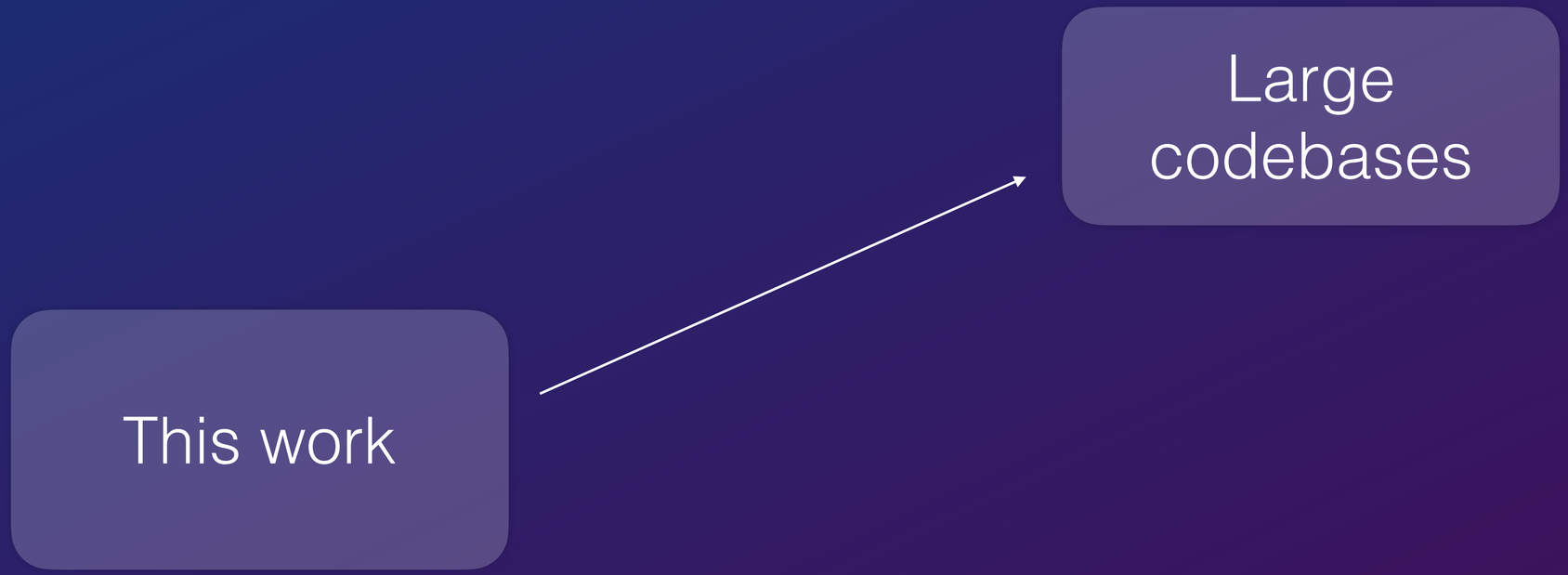
ABSTRACT

Existing program verifiers can prove advanced properties about security protocol implementations, but do not scale to large existing codebases owing to the required manual annotation overhead. We propose a methodology called Diodon that addresses this challenge by splitting the codebase into the protocol implementation (the Core) and the remainder (the Application). This split allows us to focus on applying powerful but expensive verification techniques to the security-critical Core, whereas an automatic static analysis ensures that the Application is *not* security-critical in the sense that it cannot invalidate the security properties proved for the Core. The static analysis achieves that by proving *I/O independence*, i.e., that the I/O operations within the Application are independent of the Core's security-relevant data (such as keys). In an instantiation of Diodon, we use the SMT-based program verifier Gobra to prove

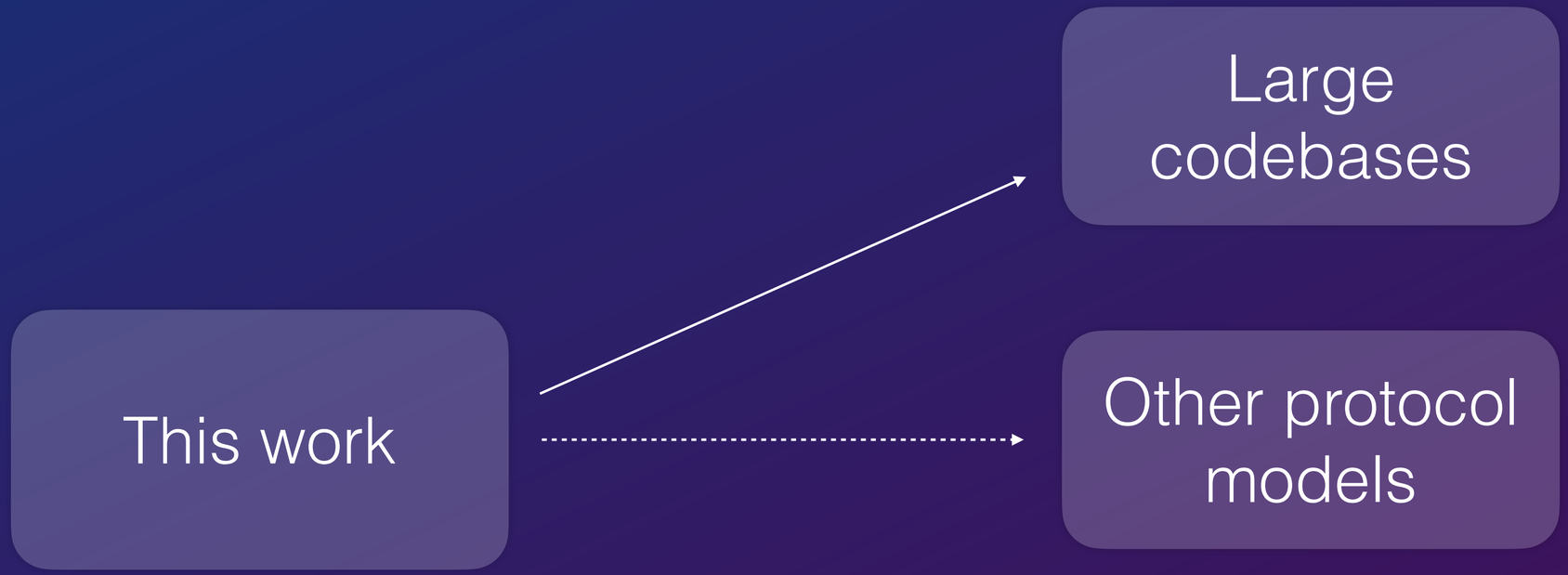
proving protocol *models* secure does not result in secure *implementations* on its own. Coding errors such as missing bounds checks (e.g., causing the Heartbleed [27] bug), omitted protocol steps (as in the Matrix SDK [28]), or ignored errors (e.g., returned by a TLS library [29, 30]) may invalidate all security properties proven for the corresponding model.

Verification of security properties for protocol *implementations* is possible [7, 8, 32]. For instance, Arquint et al. [8] first verify security properties for a Tamarin model of the protocol in the presence of a Dolev-Yao (DY) attacker [31] fully controlling the network. Then, they prove that the protocol implementation refines this model, i.e., that each I/O operation performed in the implementation is justified by the model. Refinement guarantees that the implementation inherits the security properties proven for the model. However, existing approaches to verifying protocol implementations are sound

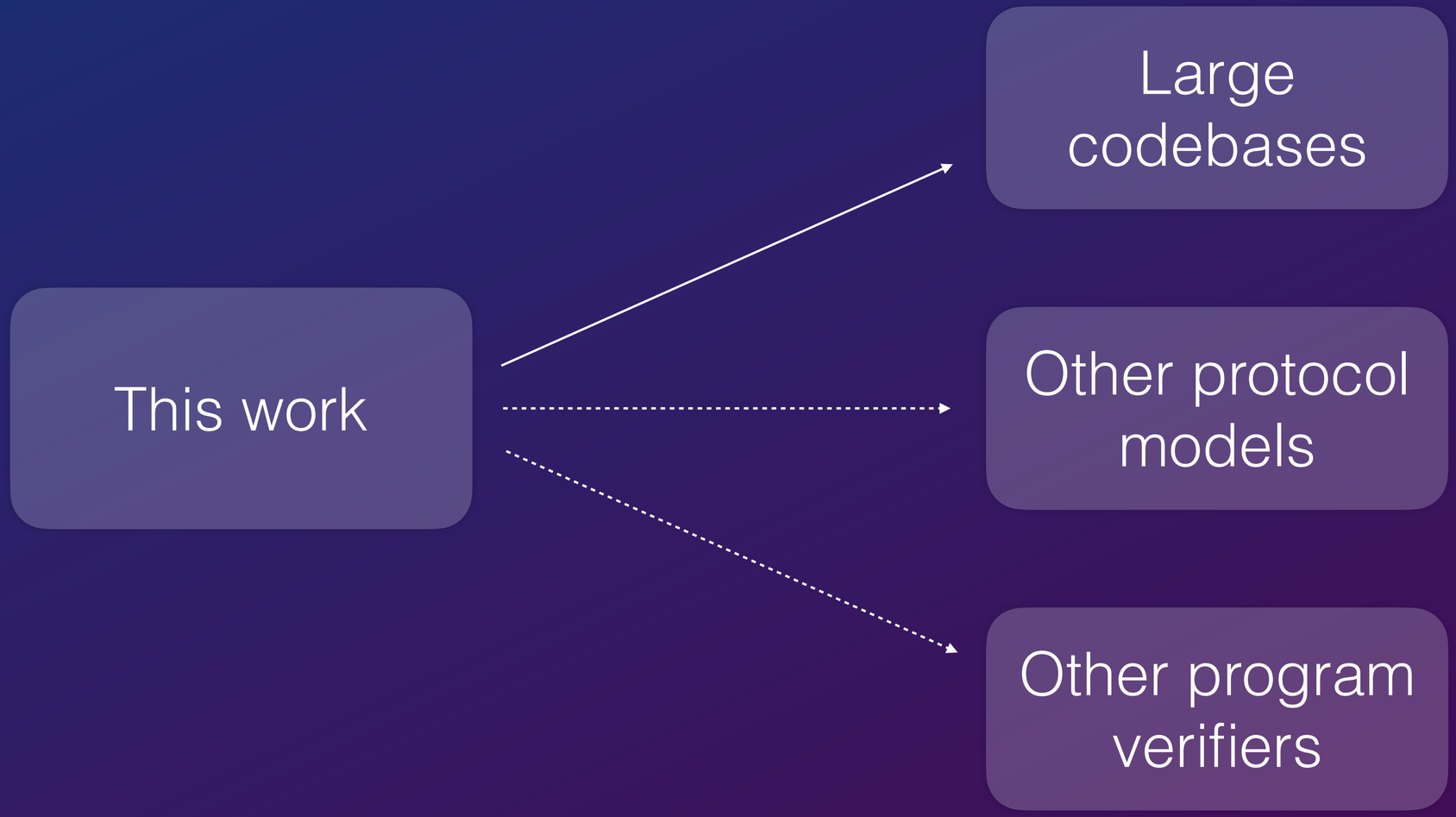
Extensions



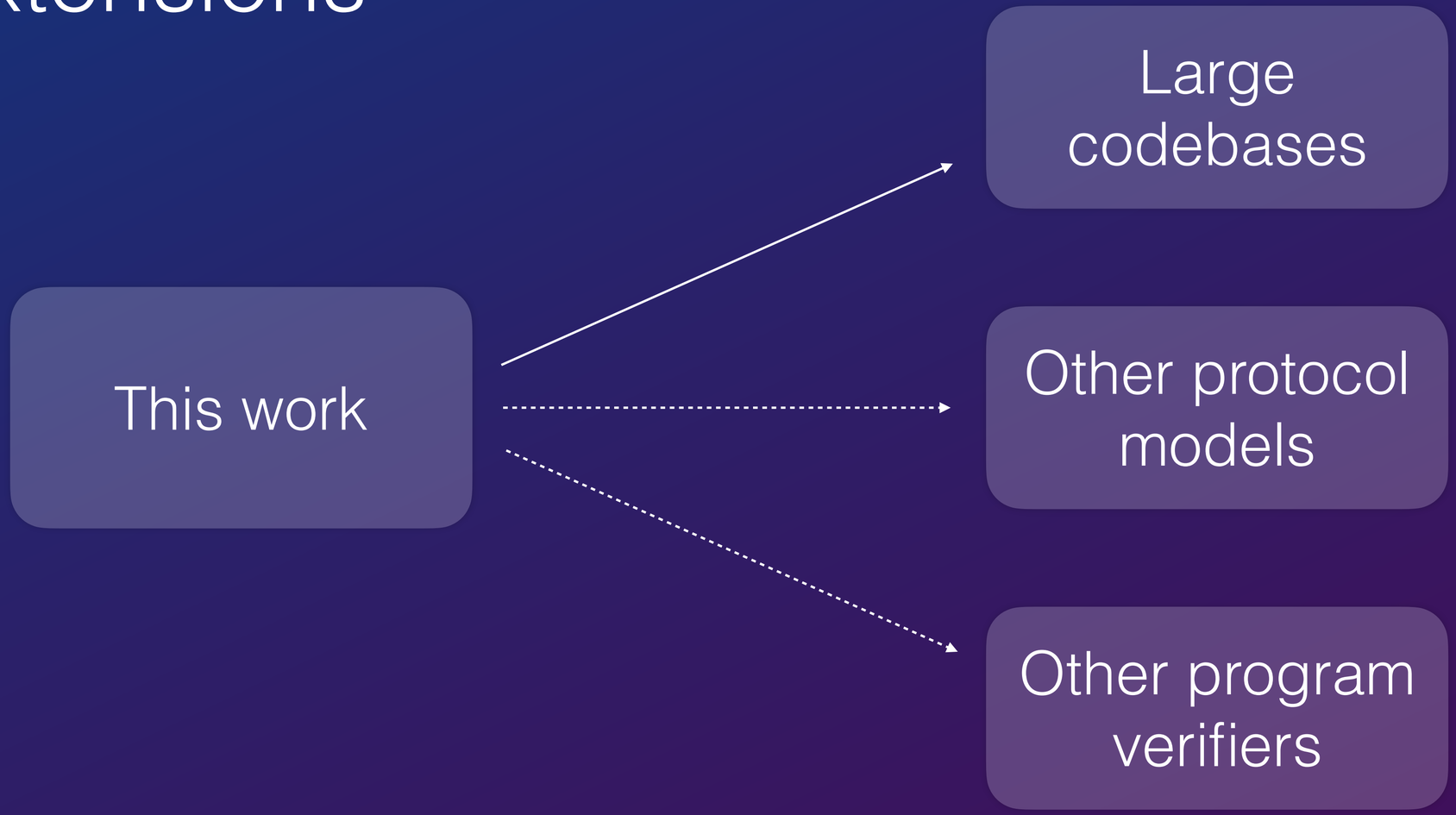
Extensions



Extensions


This work
Large codebases
Other protocol models
Other program verifiers

Extensions



compositional

Conclusions

Verification of
Protocol Models
in Tamarin

Verification of
Implementations
in Program Verifiers

Sound end-to-end verification

S&P ’23:
Sound Verification of Security Protocols:
From Design to Interoperable Implementations

Automatic specification generation

Soundness Proof &
Novel approach to relate
symbolic terms and bytes

Conclusions

Verification of
Protocol Models
in Tamarin

Verification of
Implementations
in Program Verifiers

Sound end-to-end verification

S&P ’23:
Sound Verification of Security Protocols:
From Design to Interoperable Implementations

Automatic specification generation

Soundness Proof &
Novel approach to relate
symbolic terms and bytes