CENTRE DE RECHERCHE COMMUN
INRIA
MICROSOFT RESEARCH

Secure Distributed Computations (and their Proofs)

Pedro Adao
Gilles Barthe
Ricardo Corin
Pierre-Malo Deniélou
Gurvan le Guernic
Nataliya Guts
Eugen Zalinescu
Santiago Zanella Béguelin
Karthik Bhargavan
Cédric Fournet
Benjamin Grégoire
James J. Leifer
Jérémy Planul
Tamara Rezk
Francesco Zappa Nardelli
Bruno Blanchet
David Cadé
Andy Gordon
Miriam Paoila
Alfredo Pironti
Pierre-Yves Strub
Nikhil Swamy

http://www.msr-inria.inria.fr/projects/sec

Constructive Security

- Goal: enable programmers to express and enforce strong security with a reasonable amount of effort

1. We develop **verification tools**
   for programs using cryptography
   F7 CertiCrypt; also Fine ProVerif CryptoVerif

2. We build & verify **reference implementations**
   for security protocols and libraries
   WS* TLS CardSpace DKM

3. We design & prototype **security compilers**
   for secure multiparty sessions;
   for distributed information flows (this talk)

CENTRE DE RECHERCHE
COMMUN
INRIA
MICROSOFT RESEARCH

Compiling Information-Flow Security to small Trusted Computing Bases

Jérémy Planul
MSR-INRIA joint centre

Joint work with
Cédric Fournet

http://www.msr-inria.inria.fr/projects/sec/cflow

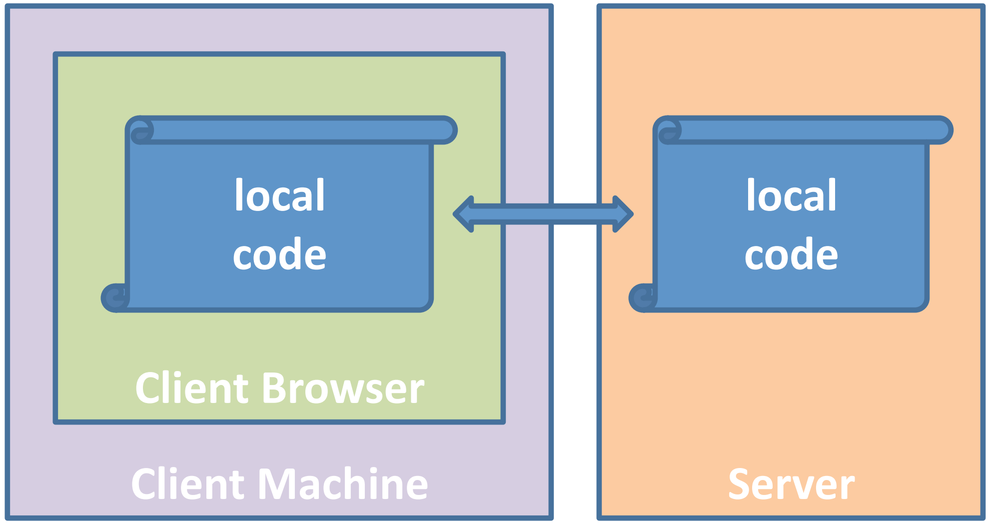
Programming with Partial Trust

- Security should hold even if the environment is partly compromised
  - Classic: the opponent controls the network
  - Modern: the opponent controls parts of the program

Programming with Partial Trust

- Security should hold even if the environment is partly compromised
  - Classic: the opponent controls the network
  - Modern: the opponent controls parts of the program

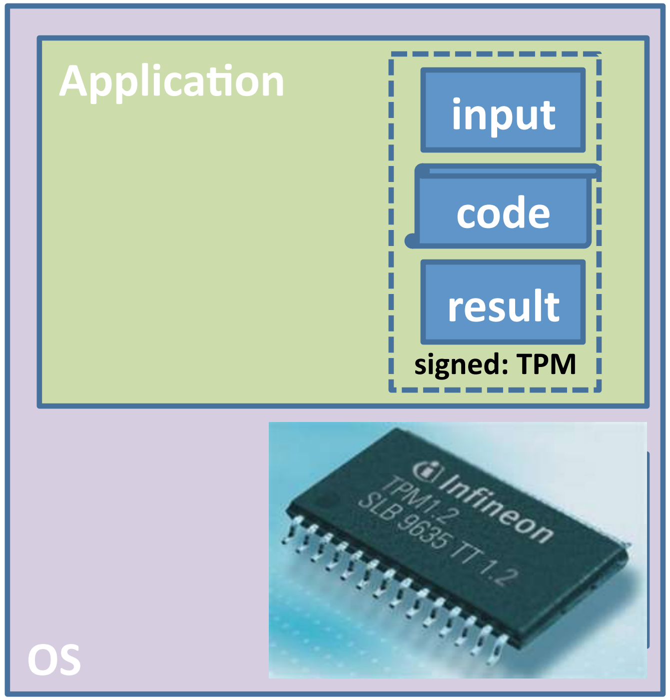
- Example: **Web Apps**
  - The network is untrusted
  - Service security should not depend on code on the client
  - Client security should not depend on services (nor their interaction)



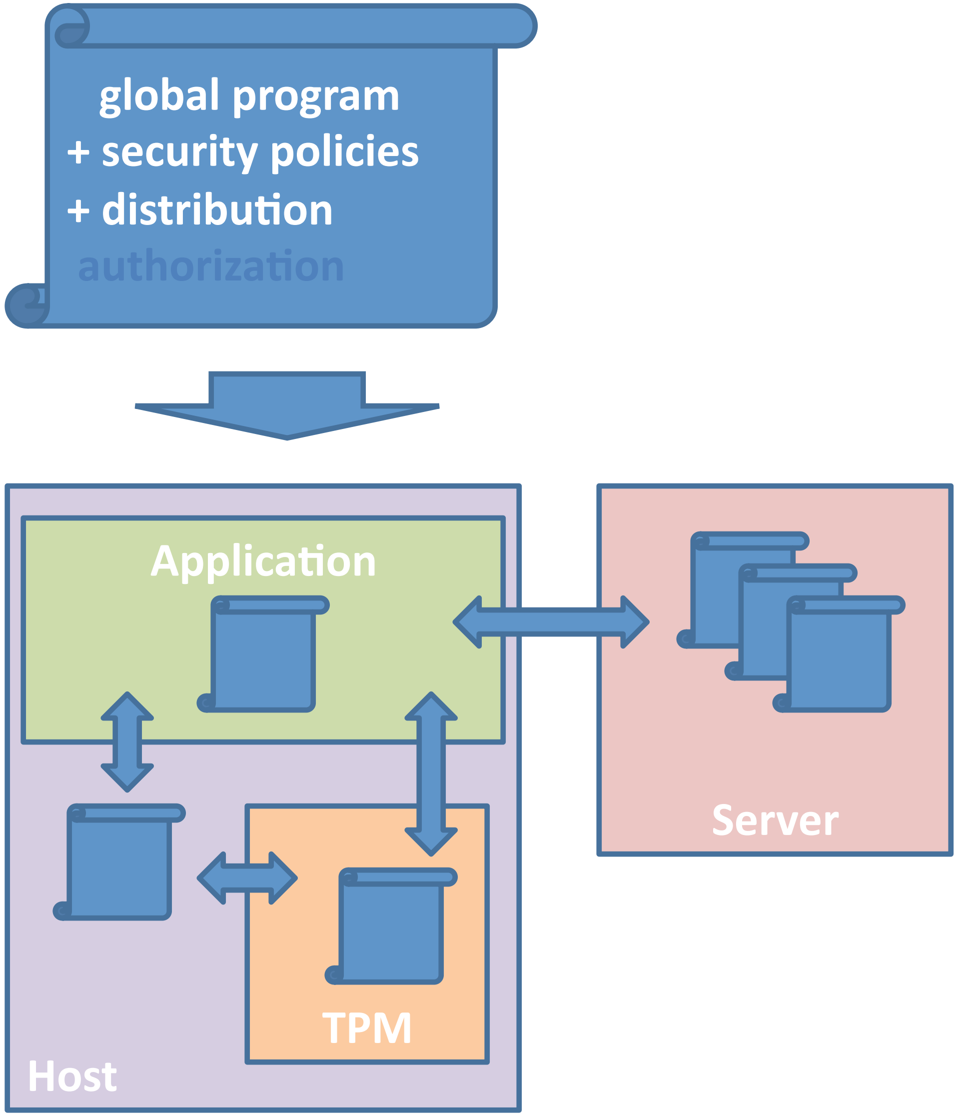
Programming with Partial Trust

- Security should hold even if the environment is partly compromised
  - Classic: the opponent controls the network
  - Modern: the opponent controls parts of the program

- A solution: **Trusted Computing**
  - Trusted Platform Modules provide HW isolation from Apps, OS, drivers,...
  - Can support secure subsystems (e.g. BitLocker)
  - Can boot short-lived kernels
  - Not used much
    No programming tools?

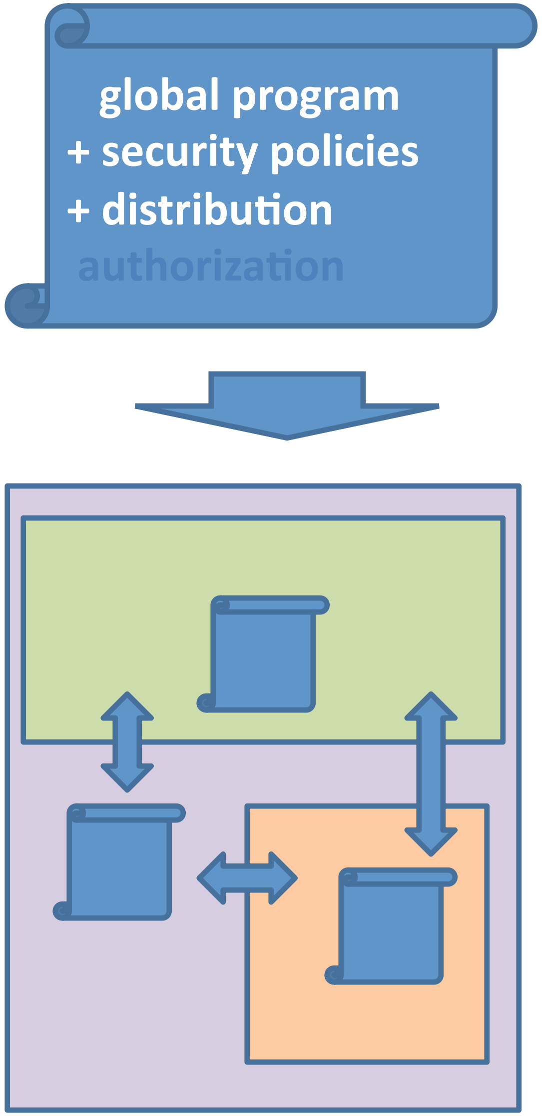


Security By Construction (Goal)



- Goal: enable programmers to express and enforce application security with a reasonable amount of effort
- We design and prototype “security compilers” that yield verified local code
- Cryptographic mechanisms are essential, and tricky

CFLOW: A Cryptographic Compiler

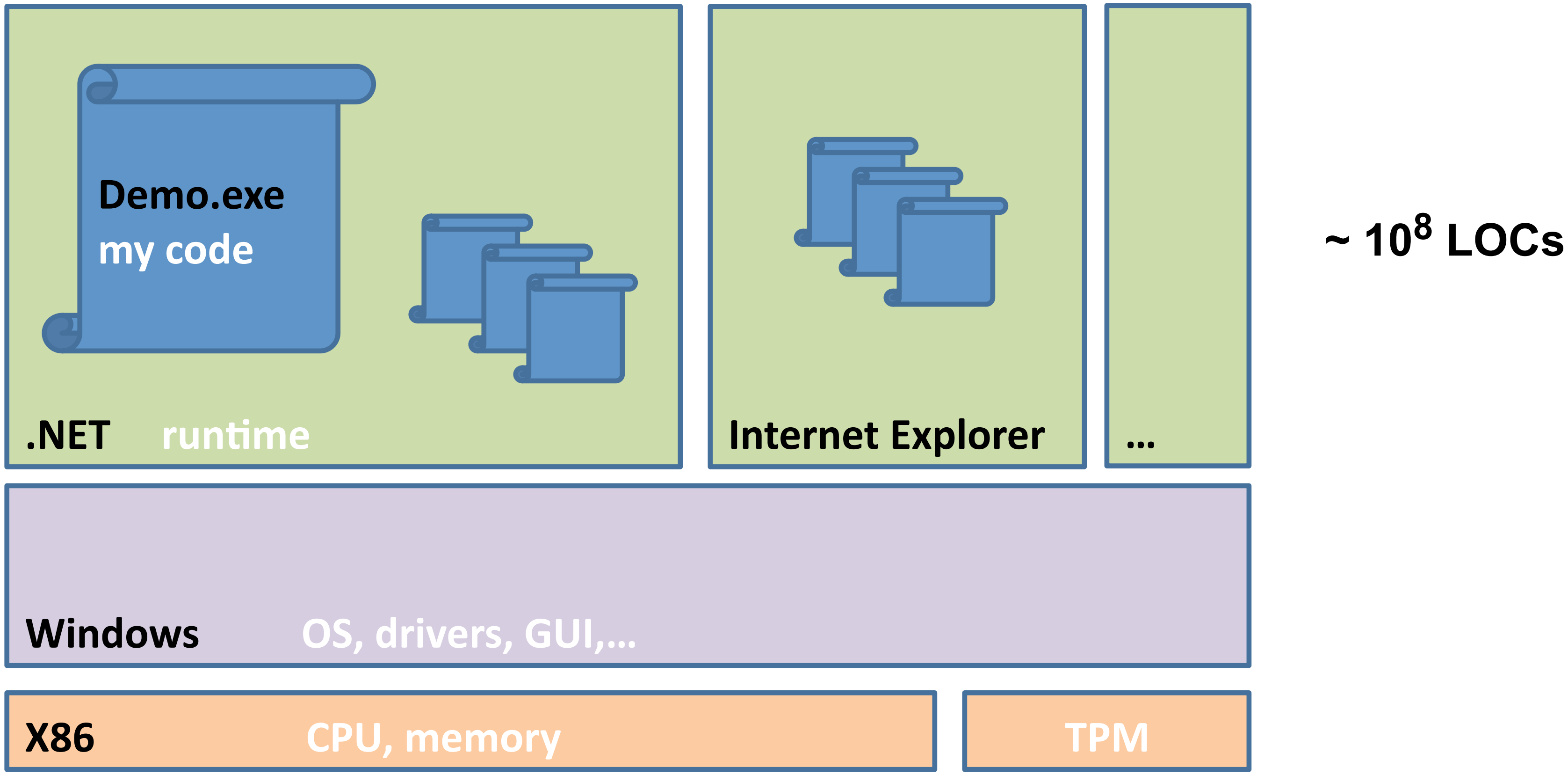


global program
+ security policies
+ distribution
authorization

1. We relate two notions of security
    - One simple and abstract, based on **information flows** in programs
    - Another more concrete, based on **cryptography & hardware assumptions** for distributed shared memory
2. We compile source programs to cryptographic distributed code
3. We show that **all source security properties are preserved**

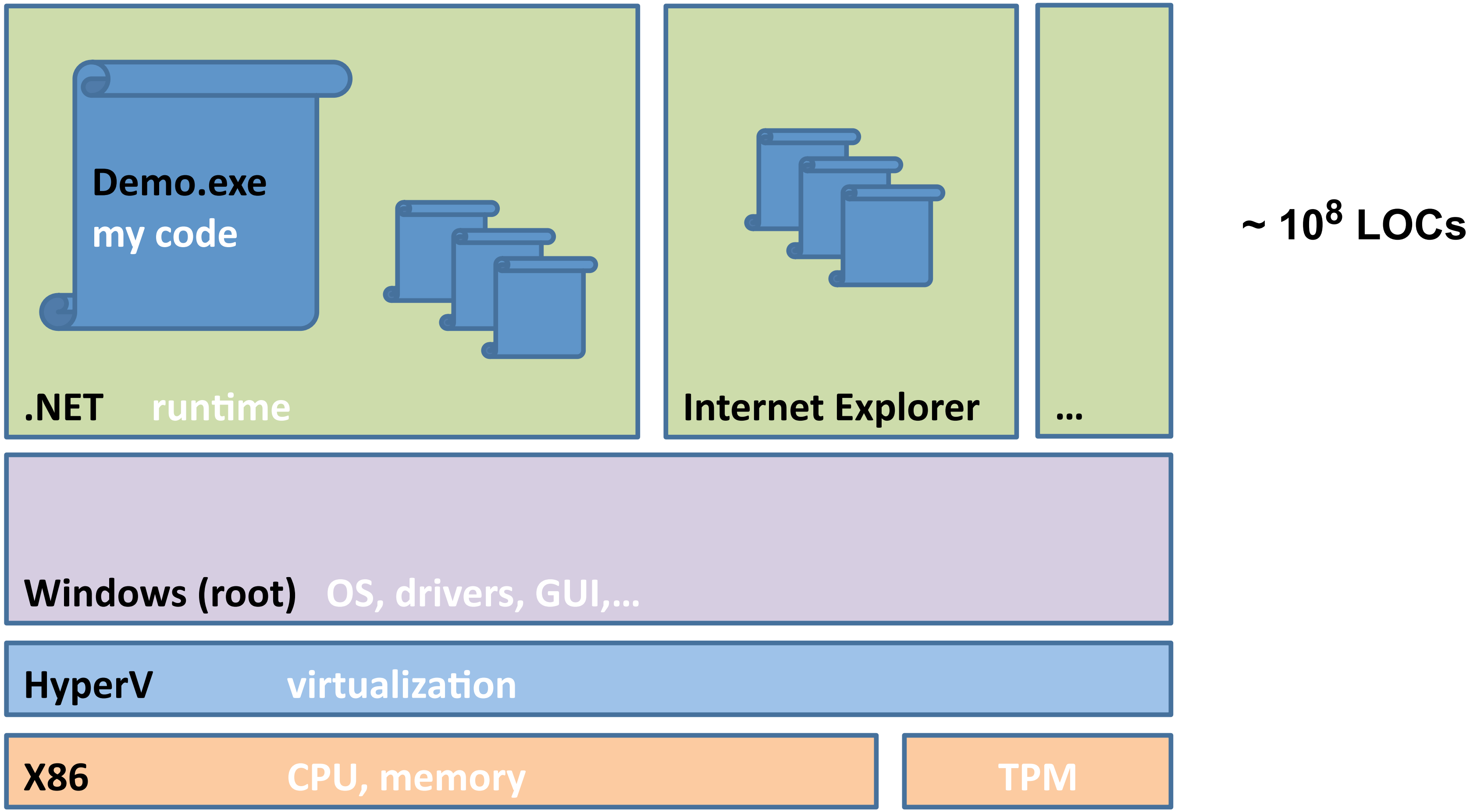
Towards Minimal TCBs

- Security should depend on a minimal trusted computing base (TCB)
  - Less critical code, easier to secure & verify



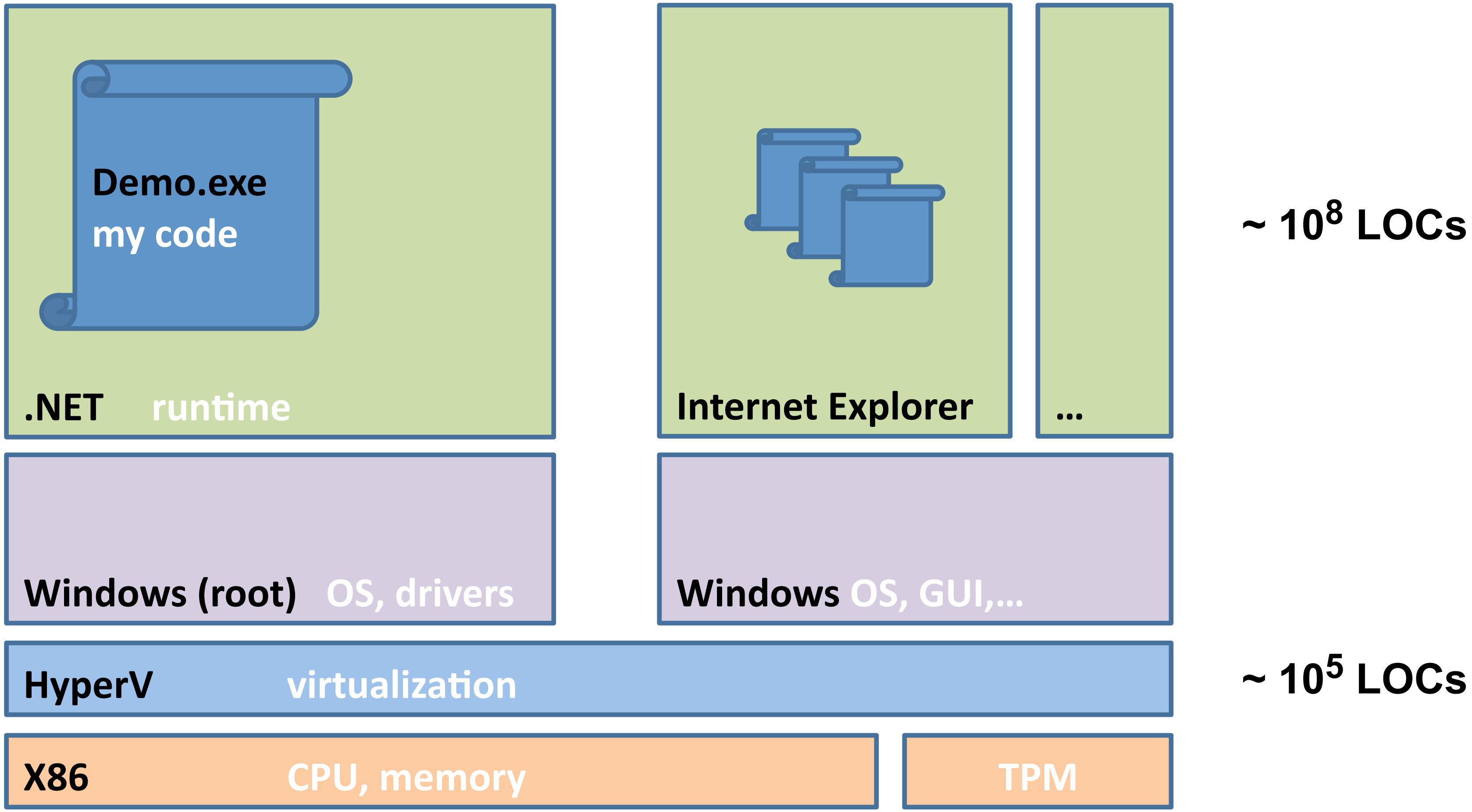
Towards Minimal TCBs

- Security should depend on a minimal trusted computing base (TCB)
  - Less critical code, easier to secure & verify



Towards Minimal TCBs

- Security should depend on a minimal trusted computing base (TCB)
  - Less critical code, easier to secure & verify



Demo.exe

my code

.NET **runtime**

Internet Explorer

…

Windows (root) **OS, drivers**

Windows **OS, GUI,…**

HyperV **virtualization**

X86 **CPU, memory**

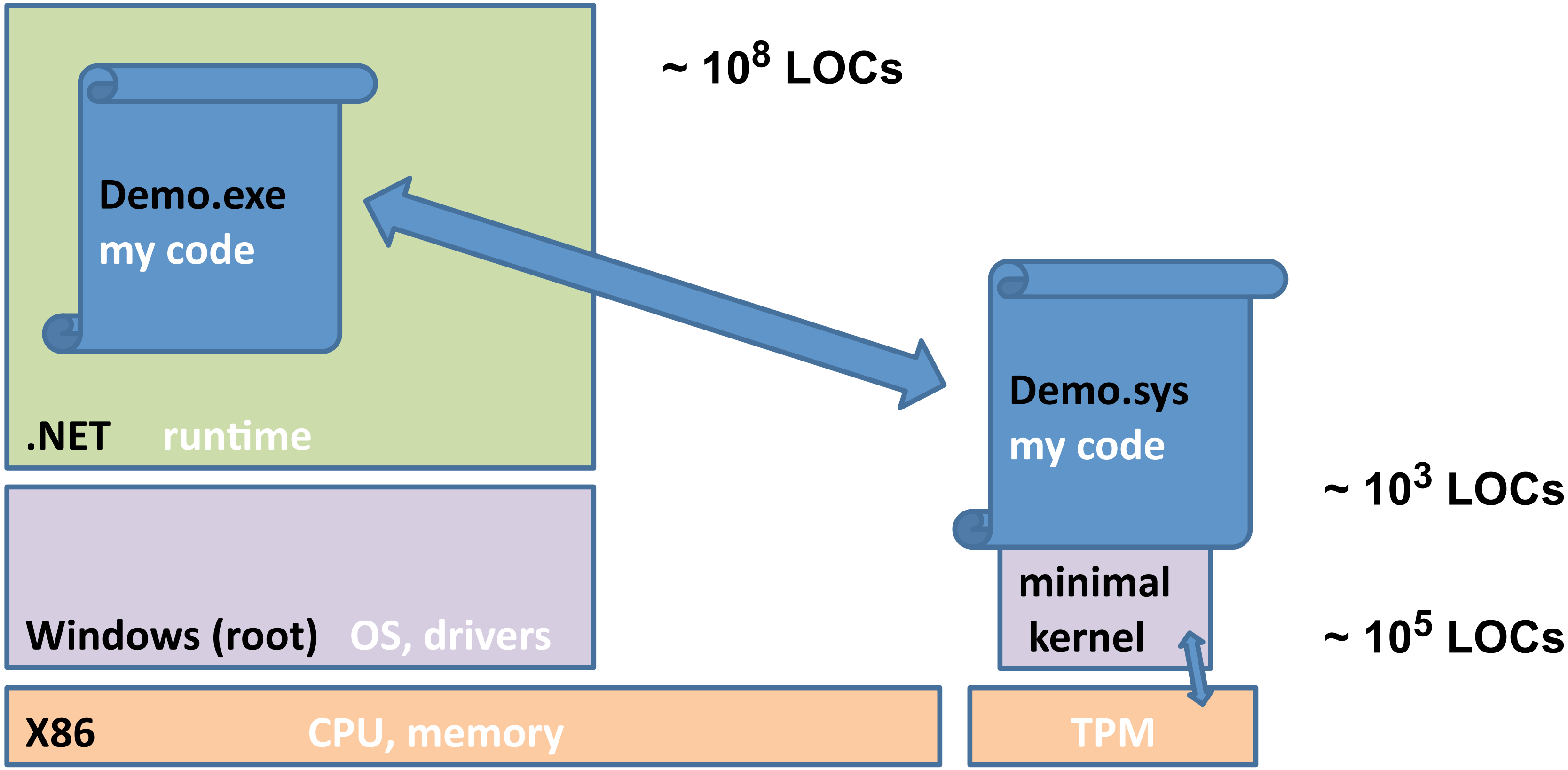
TPM

~ 10^8 LOCs

~ 10^5 LOCs

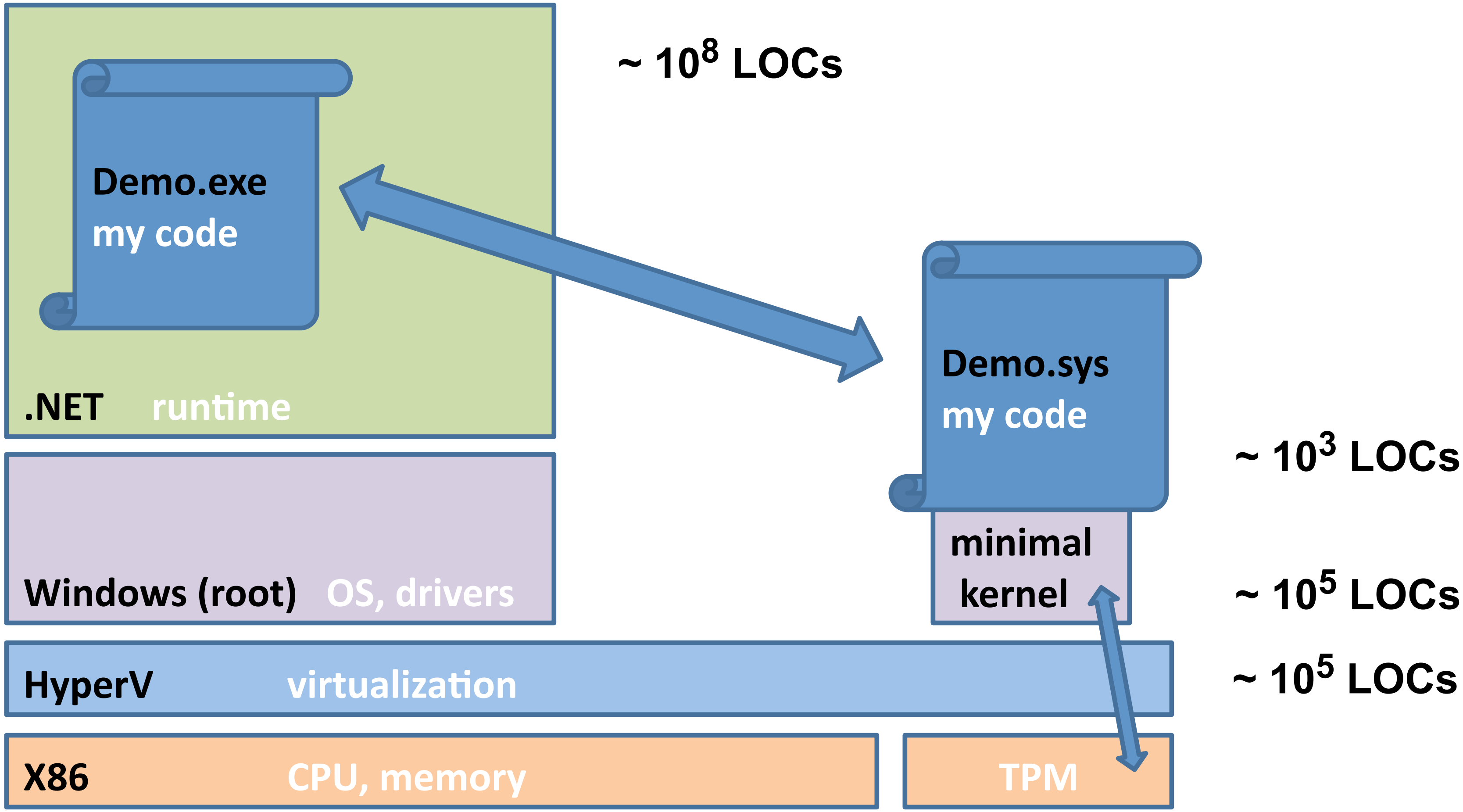
Towards Minimal TCBs

- Security should depend on a minimal trusted computing base (TCB)
  - Less critical code, easier to secure & verify



Booting Virtual Hosts with a TPM

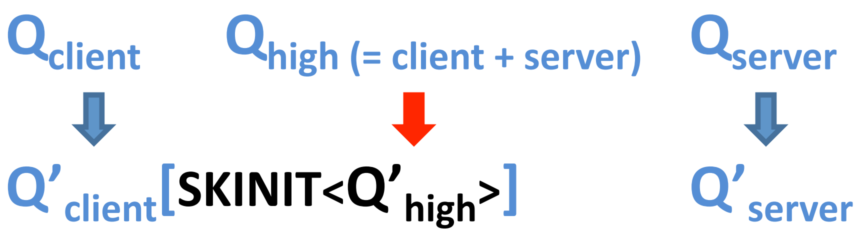
- We can compile small, secure programs for each host
- TPMs & HyperV can provide strong isolation for them (and attest it)



Booting Virtual Hosts with a TPM

1. We model TPM capabilities (this involves code as data)
2. We use CFLOW (as before) to compile programs with highly-trusted virtual hosts
3. We transform the resulting code to securely boot hosts
4. We adapt CFLOW to generate small, statically-link C code

SKINIT, SEAL, UNSEAL, EXTEND, …



Q_{client} $\quad$ $Q_{\text{high (= client + server)}}$ $\quad$ Q_{server}

$\Downarrow$ $\quad$ $\Downarrow$ $\quad$ $\Downarrow$

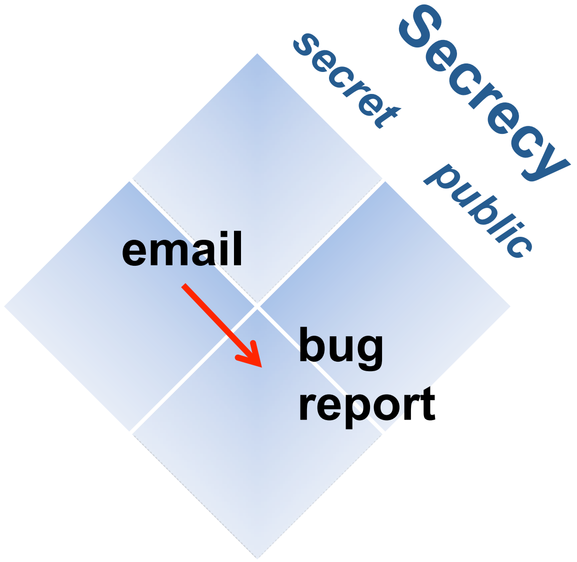
$Q'_{\text{client}}[\textbf{SKINIT}<Q'_{\text{high}}>]$ $\quad$ Q'_{server}

- Theorem: the “virtual host” transform does not enable new attacks
- We are experimenting with XCG (MSR Redmond)
  using custom builds of HyperV to run minimal TCBs

Information-Flow Security (Examples)

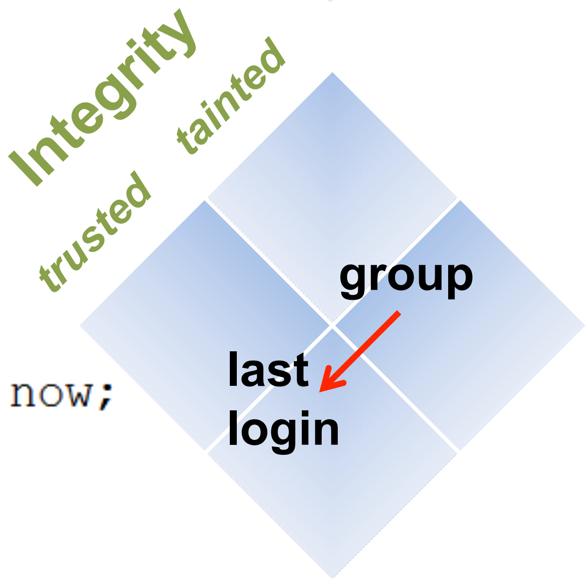
Secrecy flow
(type error)

```
bug_report := email + details;
```



Integrity flow
(type error)

```
if group = "admin" then last_login := now;
```



Cryptographic Mechanisms

- Intuitively:
  - Encryption can help preserve information secrecy
  - Signatures and MACs can help preserve information integrity

- We compose standard mechanisms, and obtain security under standard (computational) assumptions
  - Programs and adversaries are probabilistic polynomial commands
  - Correctness is relative to a security parameter, holds only with overwhelming probability
  - Proofs involve game-based reductions (simpler, more abstract models would hide crypto side channels)

Computational Cryptography

- We prove security under standard, realistic assumptions on cryptography
- Two verification approaches have been successfully applied to protocols and programs that use cryptography

  Symbolic approach (Needham-Schroeder, Dolev-Yao, ... late 70's)
  - Structural view of protocols, using formal languages and methods
  - Compositional, good tools, scales to large systems
  - Too abstract for information flows (cryptographic side channels)

  Computational approach (Yao, Goldwasser, Micali, Rivest, ... early 80's)
  - More concrete, algorithmic view
  - Adversaries range over probabilistic Turing machines
    Cryptographic materials range over bitstrings
  - More accurate, more widely accepted
  - Delicate (informal) reduction proofs; scalability issues

Computational Soundness

- We need soundness for a general class of programs so that our compiler can produce efficient code
  - We developed a type system for computational cryptography
  - We prove global correctness for each stage of the compiler
- Selected cryptographic difficulties:
  - Side channels via the usage of cryptography
    - The adversary may detect writes by observing re-encryptions
    - The adversary may detect reads by injecting bad signatures
  - Cross-dependencies between integrity and secrecy
    - Signing keys must have sufficient secrecy
    - Decryption keys must have sufficient integrity
  - Limitations on key usage (encryption cycles, key generation)
  - No information-security for keys
    - Keys need to be shared and communicated
    - Keys may be partially leaked by signing/encrypting

Cryptographic Assumptions

- Our language is expressive enough to code the algorithms, oracles, adversaries, and games of computational cryptography
  - This is the formal basis for proofs by **program transformations**

- We assume that all commands are polynomial
  - We use a global security parameter ´

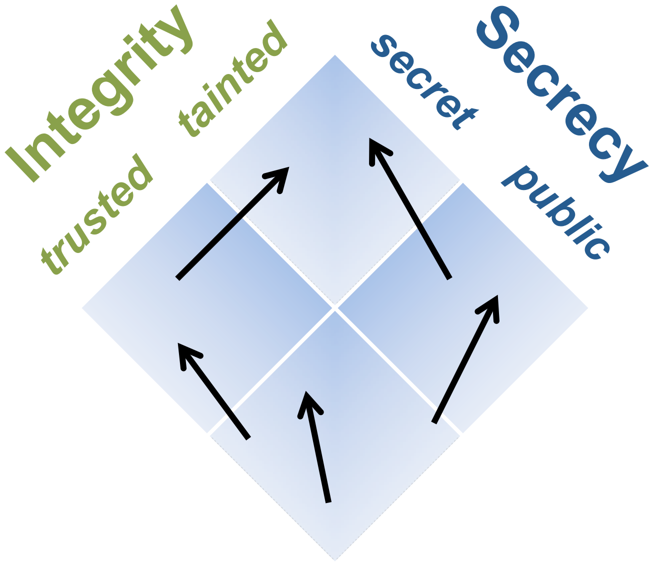
- Computational security properties are of the form
  “the command A wins a game only with negligible probability”

$$f : \mathbb{N} \to \mathbb{R} \text{ is } \textit{negligible} \text{ when } \forall c > 0, \exists n_c, \forall \eta \geq n_c, f(\eta) \leq \eta^{-c}.$$

IMPERATIVE CODE WITH INFORMATION FLOWS & DYNAMIC LINKING

Information-Flow Policies

- Security policies specify the permitted flows of information
  - Each variable has a **security level**
  - Secret variables do not leak to public variables
  - Tainted variables do not influence trusted variables



secure information flows up

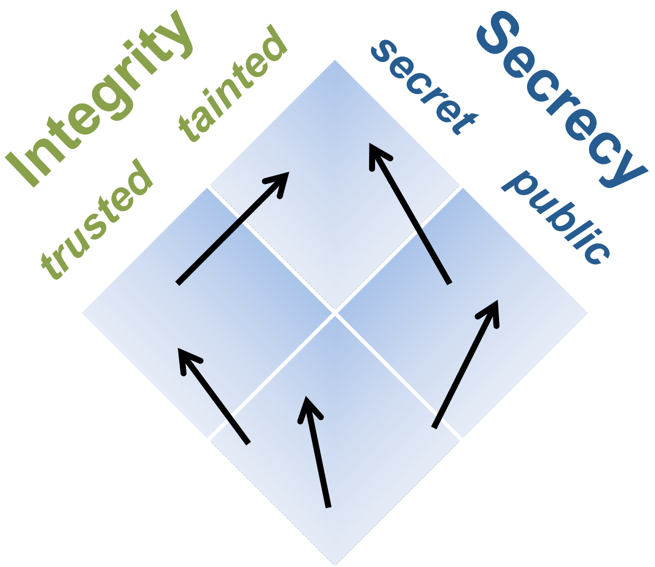
- Two complications (not for this talk)
  - Most useful programs still need to selectively **declassify** secrets and **endorse** tainted values
  - Secrecy and integrity are interdependent

This Talk

- Programming with partial trust
- Towards minimal TCBs
- Information flow security (example)
- A core language with dynamic linking
- Modelling TPM-based secure instructions
- Our compiler
    1. Generate local code for all hosts (including trusted virtual hosts)
    2. Bootstrap trusted virtual hosts using secure instructions
- Implementation examples

Information-Flow Policies

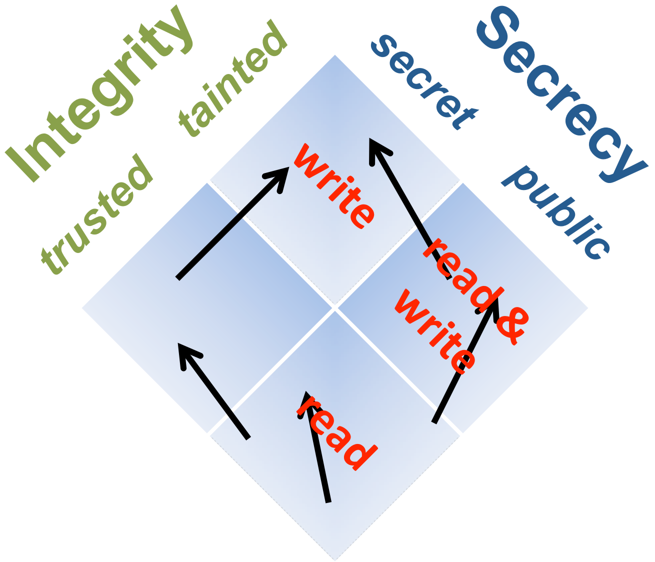
- Security levels are ordered by relative **secrecy** and **integrity**
- Security policies specify the permitted flows of information
  - Each variable has a **security level**
  - Secret variables do not leak to public variables
  - Tainted variables do not influence trusted variables



secure information flows up

Active Adversaries

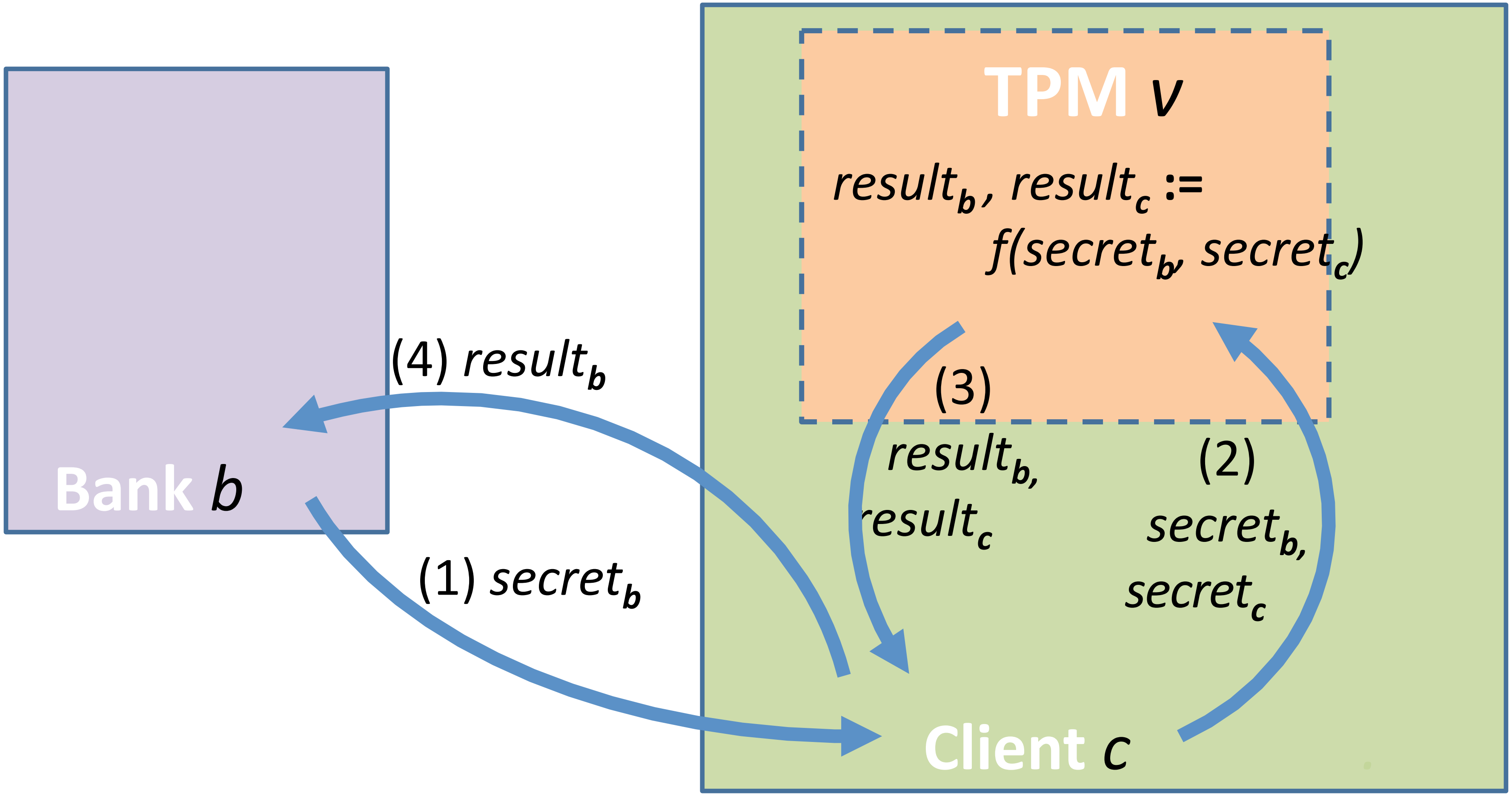
- Security levels are ordered by relative **secrecy** and **integrity**
- Security policies specify the permitted flows of information
  - Each variable has a **security level**
  - Secret variables do not leak to public variables
  - Tainted variables do not influence trusted variables
- An adversary is specified as a **compromise level**
  - Can read/write shared memory
  - Can control code at lower-level hosts



secure information flows up

Applying for a Loan (Example)

$$result_b\ ,\ result_c := f(\ secret_b\ , secret_c\ )$$



Core Language and Security Types

$$
\begin{aligned}
e &::= x \mid op(e_1, \ldots, e_n) \\
P &::= x := e \mid x := f(x_1, \ldots, x_n) \mid \mathit{skip} \mid P; P \\
&\quad \mid \text{if } e \text{ then } P \text{ else } P \mid \text{while } e \text{ do } P \mid \text{link}\, e\, [\widetilde{P}]\, \ell \mid X
\end{aligned}
$$

Core Language and Security Types

$$
\begin{aligned}
e &::= x \mid op(e_1, \ldots, e_n) \\
P &::= x := e \mid x := f(x_1, \ldots, x_n) \mid \mathit{skip} \mid P; P \\
&\quad \mid \text{if } e \text{ then } P \text{ else } P \mid \text{while } e \text{ do } P \mid \underline{\text{link } e\, [\widetilde{P}]\, \ell \mid X}
\end{aligned}
$$

$$
\text{(\textsc{TSubC})}\ \frac{\vdash P : \ell \qquad \ell' \leq \ell}{\vdash P : \ell'}
\qquad
\text{(\textsc{TFun})}\ \frac{\vdash \widetilde{y} : \Gamma(x)}{\vdash x := f(\widetilde{y}) \ : \Gamma(x)}
\qquad
\text{(\textsc{TSeq})}\ \frac{\vdash P : \ell \qquad \vdash P' : \ell}{\vdash P; P' \ : \ \ell}
\qquad
\text{(\textsc{TSkip})}\ \frac{}{\vdash \mathit{skip} : \top}
$$

$$
\text{(\textsc{TCond})}\ \frac{\vdash e : \ell \qquad \vdash P : \ell \qquad \vdash P' : \ell}{\vdash \text{if } e \text{ then } P \text{ else } P' \ : \ \ell}
\qquad
\text{(\textsc{TWhile})}\ \frac{\vdash e : \ell \qquad \vdash P \ : \ \ell}{\vdash \text{while } e \text{ do } P \ : \ \ell}
\qquad
\text{(\textsc{TVar})}\ \frac{}{\vdash X : (\bot_C, \top_I)}
$$

Strict rules:

$$
\text{(\textsc{TAssign Strict})}\ \frac{\vdash e : \Gamma(x)}{\vdash x := e : \Gamma(x)}
\qquad
\text{(\textsc{TLink Strict})}\ \frac{\vdash e : \ell \qquad \vdash \widetilde{P} : (\bot_C, \top_I)}{\vdash \text{link } e\, [\widetilde{P}]\, \ell : \ell}
$$

Lax rules:

$$
\text{(\textsc{TAssign Endorse})}\ \frac{\vdash e : (c, \_) \quad c \leq C(x)}{\vdash x := e : \Gamma(x)}
\qquad
\text{(\textsc{TAssign Robust})}\ \frac{\vdash e : (c, \_) \quad c \not\leq C(x)}{\vdash x := e : \Gamma(x) \sqcap (\top_C, R(c))}
\qquad
\text{(\textsc{TLink Privileged})}\ \frac{\vdash e : \ell \quad \vdash \widetilde{P} : \ell \quad \ell \leq \ell'}{\vdash \text{link } e\, [\widetilde{P}]\, \ell' : \ell}
$$

Core Language and Security Types

$$
\begin{aligned}
e &::= x \mid op(e_1, \ldots, e_n) \\
P &::= x := e \mid x := f(x_1, \ldots, x_n) \mid \mathit{skip} \mid P; P \\
&\quad \mid \text{if } e \text{ then } P \text{ else } P \mid \text{while } e \text{ do } P \mid \text{link } e\,[\widetilde{P}]\,\ell \mid X
\end{aligned}
$$

(TSubC)
$$\frac{\vdash P : \ell \qquad \ell' \leq \ell}{\vdash P : \ell'}$$

(TFun)
$$\frac{\vdash \tilde{y} : \Gamma(x)}{\vdash x := f(\tilde{y}) \ : \Gamma(x)}$$

(TSeq)
$$\frac{\vdash P : \ell \qquad \vdash P' : \ell}{\vdash P; P' \ : \ell}$$

(TSkip)
$$\frac{}{\vdash \mathit{skip} : \top}$$

(TCond)
$$\frac{\vdash e : \ell \qquad \vdash P : \ell \qquad \vdash P' : \ell}{\vdash \text{if } e \text{ then } P \text{ else } P' \ : \ \ell}$$

(TWhile)
$$\frac{\vdash e : \ell \qquad \vdash P \ : \ \ell}{\vdash \text{while } e \text{ do } P \ : \ \ell}$$

(TVar)
$$\frac{}{\vdash X : (\bot_C, \top_I)}$$

Strict rules:

(TAssign Strict)
$$\frac{\vdash e : \Gamma(x)}{\vdash x := e : \Gamma(x)}$$

+

(TLink Strict)
$$\frac{\vdash e : \ell \qquad \vdash \widetilde{P} : (\bot_C, \top_I)}{\vdash \text{link } e\,[\widetilde{P}]\,\ell : \ell}$$

Lax rules:

(TAssign Endorse)
$$\frac{\vdash e : (c, \_) \qquad c \leq C(x)}{\vdash x := e : \Gamma(x)}$$

(TAssign Robust)
$$\frac{\vdash e : (c, \_) \qquad c \not\leq C(x)}{\vdash x := e : \Gamma(x) \sqcap (\top_C, R(c))}$$

(TLink Privileged)
$$\frac{\vdash e : \ell \qquad \vdash \widetilde{P} : \ell \qquad \ell \leq \ell'}{\vdash \text{link } e\,[\widetilde{P}]\,\ell' : \ell}$$

Core Language and Security Types

$$
\begin{aligned}
e &::= x \mid op(e_1, \ldots, e_n) \\
P &::= x := e \mid x := f(x_1, \ldots, x_n) \mid \mathit{skip} \mid P; P \\
&\quad \mid \text{if } e \text{ then } P \text{ else } P \mid \text{while } e \text{ do } P \mid \text{link}\, e\, [\widetilde{P}]\, \ell \mid X
\end{aligned}
$$

$$
(\textsc{TSubC})\ \frac{\vdash P : \ell \qquad \ell' \leq \ell}{\vdash P : \ell'}
\qquad
(\textsc{TFun})\ \frac{\vdash \tilde{y} : \Gamma(x)}{\vdash x := f(\tilde{y}) \ : \Gamma(x)}
\qquad
(\textsc{TSeq})\ \frac{\vdash P : \ell \qquad \vdash P' : \ell}{\vdash P; P' \ : \ell}
\qquad
(\textsc{TSkip})\ \frac{}{\vdash \mathit{skip} : \top}
$$

$$
(\textsc{TCond})\ \frac{\vdash e : \ell \qquad \vdash P : \ell \qquad \vdash P' : \ell}{\vdash \text{if } e \text{ then } P \text{ else } P' \ : \ \ell}
\qquad
(\textsc{TWhile})\ \frac{\vdash e : \ell \qquad \vdash P \ : \ \ell}{\vdash \text{while } e \text{ do } P \ : \ \ell}
\qquad
(\textsc{TVar})\ \frac{}{\vdash X : (\bot_C, \top_I)}
$$

Strict rules:

$$
(\textsc{TAssign Strict})\ \frac{\vdash e : \Gamma(x)}{\vdash x := e : \Gamma(x)}
\qquad
(\textsc{TLink Strict})\ \frac{\vdash e : \ell \qquad \vdash \widetilde{P} : (\bot_C, \top_I)}{\vdash \text{link}\, e\, [\widetilde{P}]\, \ell : \ell}
$$

Lax rules:

$$
(\textsc{TAssign Endorse})\ \frac{\vdash e : (c, \_) \qquad c \leq C(x)}{\vdash x := e : \Gamma(x)}
\qquad
(\textsc{TAssign Robust})\ \frac{\vdash e : (c, \_) \qquad c \not\leq C(x)}{\vdash x := e : \Gamma(x) \sqcap (\top_C, R(c))}
\qquad
(\textsc{TLink Privileged})\ \frac{\vdash e : \ell \qquad \vdash \widetilde{P} : \ell \qquad \ell \leq \ell'}{\vdash \text{link}\, e\, [\widetilde{P}]\, \ell' : \ell}
$$

Security Types

- ¡

Linking with Privileged Code

- The link command

$$\frac{[\![e]\!](\mu) = \langle P \rangle \qquad \vdash P : \ell}{\langle \text{link}\; e\; [\widetilde{P}]\; \ell, \mu \rangle \rightsquigarrow_1 \langle P[\widetilde{P}/\widetilde{X}], \mu \rangle}$$

  - Turns data (*e*) into executable code (*P*)
  - Dynamically checks that code
  - Runs that code linked with subcommands

- Example: PIN-based access control

$$\begin{array}{l} c := 0; \\ \text{link}\; a[\text{if}\; c < 3\; \&\&\; \mathit{guess} = \mathit{pwd}\; \text{then}\; r := \mathit{secret}\; \text{else}\; c{+}{+}]\; \mathit{LL} \end{array}$$

  - The adversary can read/write *a, guess, r* but not *pwd, c, secret*
  - The command runs adversary code,
    which may try to guess the password at most three 3 times.

TPM-Based Secure Instructions (1/2)

- Monotonic Counters (for linearity)

$$\text{INC} \doteq \quad c := c+1 \qquad\qquad \mathit{\Gamma}(c) = \ell^{I}_{\mathit{TPM}}$$

- Platform Configuration Registers (for boot integrity)

$$\text{EXTEND}_i \doteq \quad h_i := \mathcal{H}(h_i | \mathit{identity}) \qquad \mathit{\Gamma}(h_i) = \ell^{I}_{\mathit{TPM}}$$

- Secure Late Boots (for small, short-lived kernels)

$$\text{SKINIT} \doteq \quad h_{17} := \mathcal{H}(kernel); \text{link } \mathit{kernel}[\widetilde{\mathit{TPM}}]\ \ell^{I}_{system}; h_{17} := 0$$

- Remote Attestation
- Sealing & unsealing

TPM-Based Secure Instructions (2/2)

- Monotonic Counters (for linearity)
- Platform Configuration Registers (for boot integrity)
- Secure Late Boots (for small, short-lived kernels)
- Remote Attestation (for the current configuration)

$$\begin{aligned}
\mathrm{ATTEST}_i &\doteq \quad \mathit{tag} := \mathcal{S}(i|h_i|\mathit{plain},\, k^-_{\mathit{TPM}}) \\
\mathrm{VERIFY}_i &\doteq \quad \text{if } \mathcal{V}(i|\mathit{source}|\mathit{plain},\, \mathit{tag},\, k^+_{\mathit{TPM}}) \text{ then } X
\end{aligned}$$

- Sealing & unsealing (for keeping private state)

$$\begin{aligned}
\mathrm{SEAL}_i &\doteq \quad \mathit{enc} := \mathcal{SE}(\mathit{plain}, s.\mathit{ke});\ \mathit{mac} := \mathcal{M}(i|h_i|\mathit{target}|\mathit{enc}, s.\mathit{ka}); \\
& \quad \mathit{cipher} := \mathit{enc}|\mathit{mac};\ \mathit{enc} := 0;\ \mathit{mac} := 0 \\
\mathrm{UNSEAL}_i &\doteq \quad \mathit{enc}|\mathit{mac} := \mathit{cipher}; \\
& \quad \text{if } \mathcal{V}_{\mathcal{M}}(i|\mathit{source}|h_i|\mathit{enc},\, \mathit{mac},\, s.\mathit{ka}) \\
& \quad \text{then } \mathit{plain} := \mathcal{SD}(\mathit{enc}, s.\mathit{ke}) \text{ else } \mathit{plain} := 0; \\
& \quad \mathit{enc} := 0;\ \mathit{mac} := 0
\end{aligned}$$

Probabilistic Semantics

- A standard WHILE language with shared memory

$$\begin{aligned} e &::= x \mid v \mid op(e_1, \ldots, e_n) \\ P &::= x := e \mid x := f(x_1, \ldots, x_n) \mid \\ &\qquad P; P \mid \textsf{if } e \textsf{ then } P \textsf{ else } P \mid \textsf{while } e \textsf{ do } P \mid \textsf{skip} \end{aligned}$$

- A probabilistic semantics between configurations (Markov chains)

$$\langle P, \mu \rangle \longrightarrow_p \langle P', \mu' \rangle$$

  - so programs can represent cryptographic algorithms

Probabilistic Semantics

- A standard WHILE language with shared memory

$$\begin{aligned} e &::= x \mid v \mid op(e_1, \ldots, e_n) \\ P &::= x := e \mid x := f(x_1, \ldots, x_n) \mid \\ &\qquad P; P \mid \textsf{if } e \textsf{ then } P \textsf{ else } P \mid \textsf{while } e \textsf{ do } P \mid \textsf{skip} \end{aligned}$$

- A probabilistic semantics between configurations (Markov chains)

$$\langle P, \mu \rangle \longrightarrow_p \langle P', \mu' \rangle$$

  - so programs can represent cryptographic algorithms
  - in particular, we use a "fair coin-tossing" function:

$$\langle x := \{0, 1\}; P, \mu \rangle \longrightarrow_{\frac{1}{2}} \langle P, \mu[x = b] \rangle \text{ for } b = 0, 1$$

Probabilistic Semantics

ASSIGNS

$$\frac{[\![e]\!](\mu) = v}{\langle x := e, \mu \rangle \rightsquigarrow_1 \langle \surd, \mu\{x \mapsto v\} \rangle}$$

SEQS

$$\frac{\langle P, \mu \rangle \rightsquigarrow_p \langle P_1, \mu_1 \rangle \qquad P_1 \neq \surd}{\langle P; P', \mu \rangle \rightsquigarrow_p \langle P_1; P', \mu_1 \rangle}$$

SEQT

$$\frac{\langle P, \mu \rangle \rightsquigarrow_p \langle \surd, \mu_1 \rangle}{\langle P; P', \mu \rangle \rightsquigarrow_p \langle P', \mu_1 \rangle}$$

SKIPS

$$\langle \textsf{skip}, \mu \rangle \rightsquigarrow_1 \langle \surd, \mu \rangle$$

STABLE

$$\langle \surd, \mu \rangle \rightsquigarrow_1 \langle \surd, \mu \rangle$$

CONDTRUE

$$\frac{[\![e]\!](\mu) = \textsf{true}}{\langle \textsf{if } e \textsf{ then } P \textsf{ else } P', \mu \rangle \rightsquigarrow_1 \langle P, \mu \rangle}$$

CONDFALSE

$$\frac{[\![e]\!](\mu) \neq \textsf{true}}{\langle \textsf{if } e \textsf{ then } P \textsf{ else } P', \mu \rangle \rightsquigarrow_1 \langle P', \mu \rangle}$$

WHILETRUE

$$\frac{[\![e]\!](\mu) = \textsf{true}}{\langle \textsf{while } e \textsf{ do } P, \mu \rangle \rightsquigarrow_1 \langle P; \textsf{while } e \textsf{ do } P, \mu \rangle}$$

WHILEFALSE

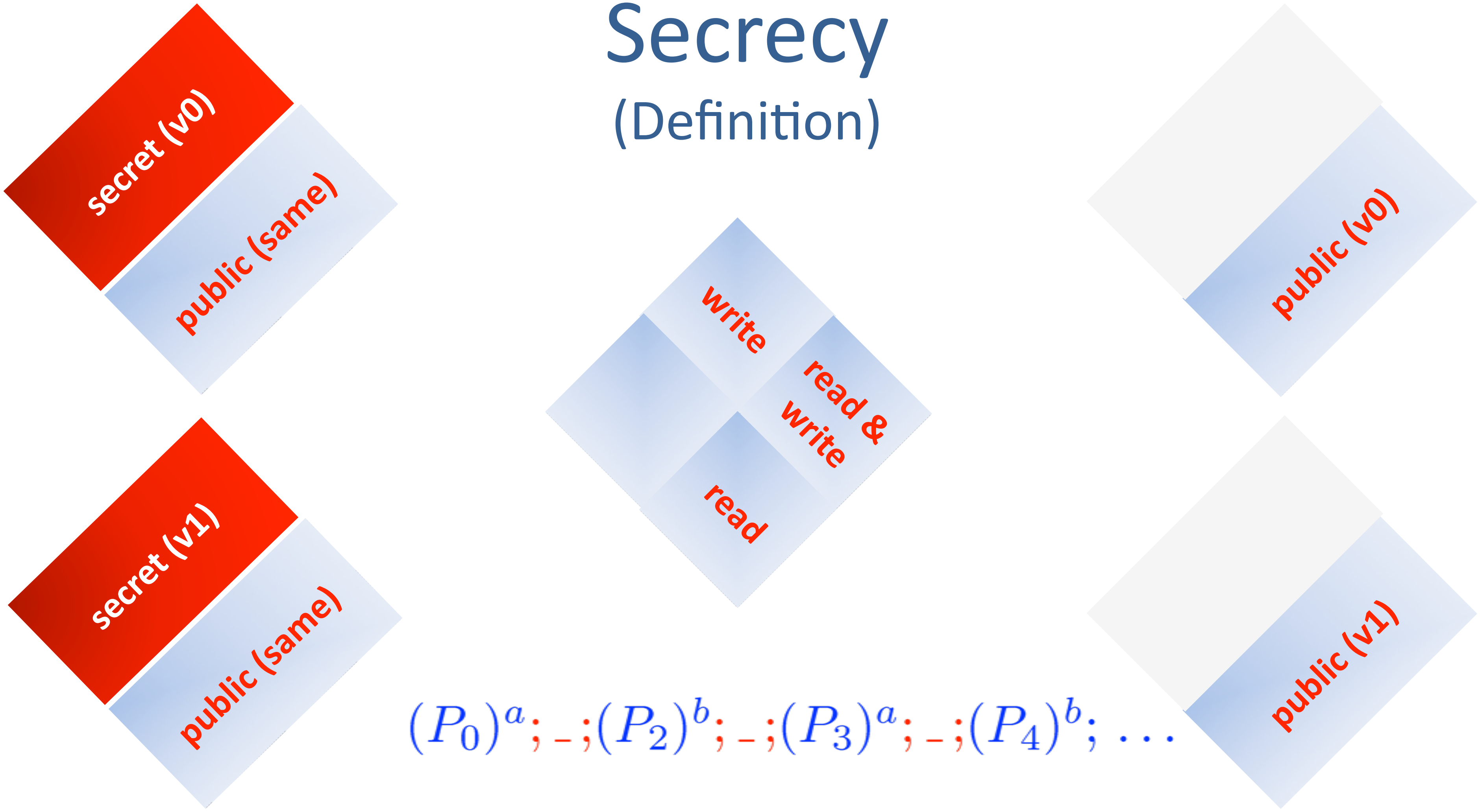
$$\frac{[\![e]\!](\mu) \neq \textsf{true}}{\langle \textsf{while } e \textsf{ do } P, \mu \rangle \rightsquigarrow_1 \langle \surd, \mu \rangle}$$

FUN

$$\frac{p = [\![f]\!](\mu(y_1), \ldots, \mu(y_n))(\vec{v}) \qquad p > 0}{\langle \vec{x} := f(y_1, \ldots, y_n), \mu \rangle \rightsquigarrow_p \langle \surd, \mu\{\vec{x} \mapsto \vec{v}\} \rangle}$$

Secrecy

(Definition)



$$(P_0)^a; \_; (P_2)^b; \_; (P_3)^a; \_; (P_4)^b; \ldots$$

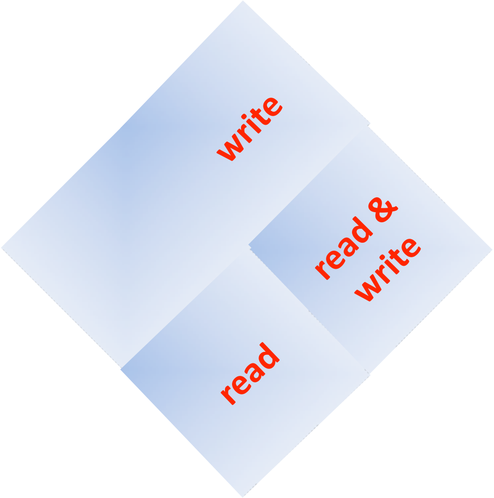
1. **Pick two initial memories with same public values**
2. **Run (twice) the same program interleaved with any adversary code**
3. **Do we obtain the same public results?**

Computational Secrecy Games (Definition)

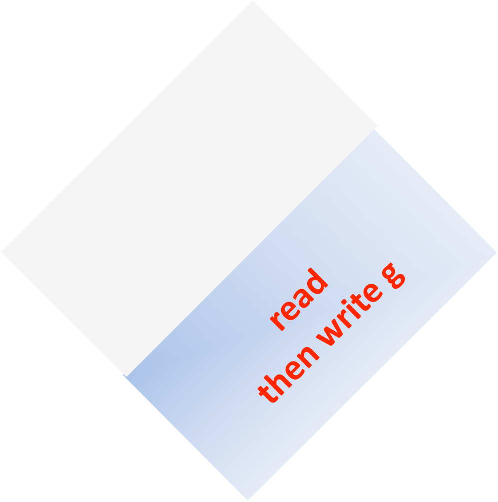
For all commands I, B_0, B_1, $\vec{A}$, T (...), run

$b \leftarrow \{0, 1\}; I;$
if b then B_0 else B_1;

$P[\vec{A}];$



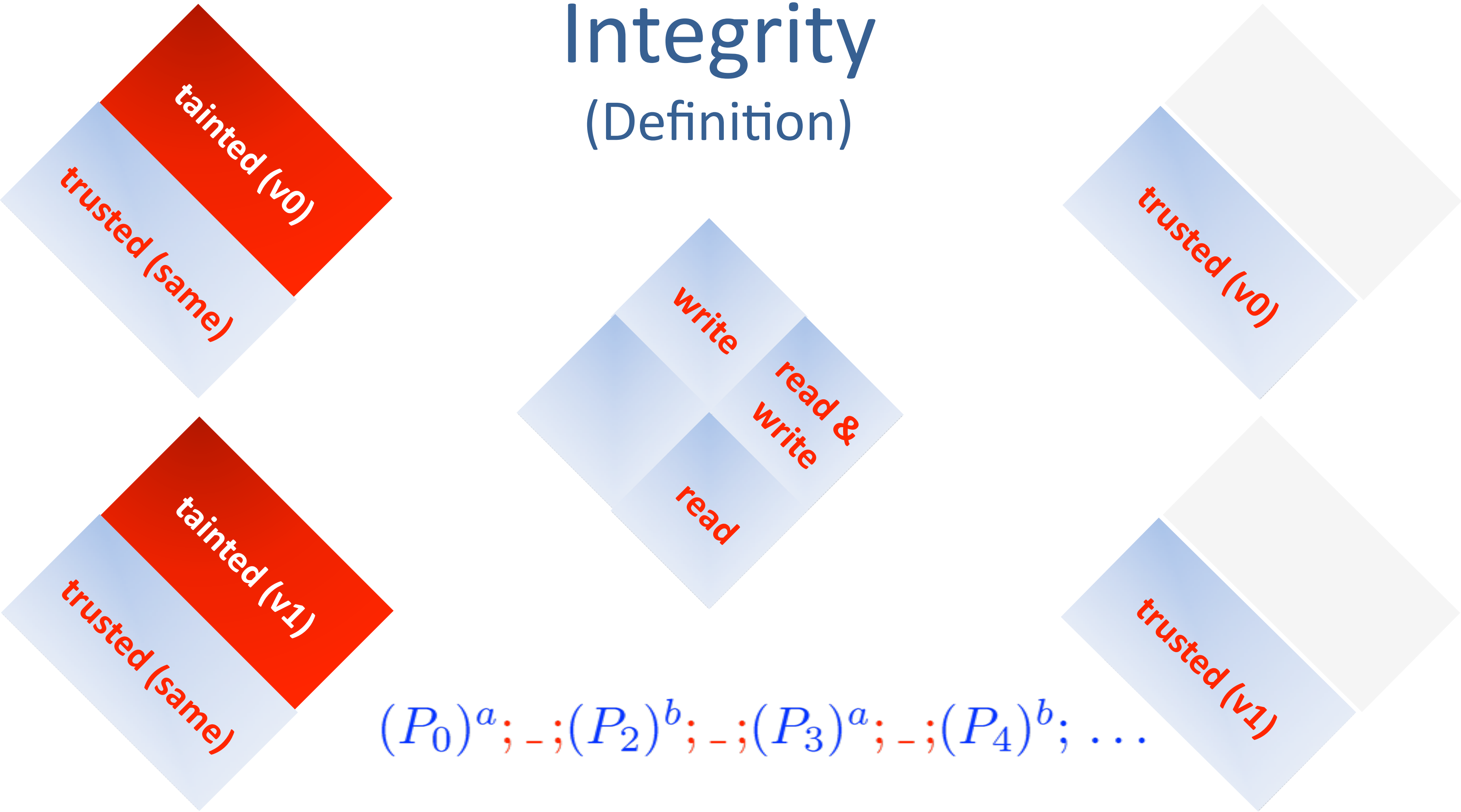
T



P preserves secrecy when $Pr(b = g) \le \frac{1}{2} + \epsilon(\eta)$

Integrity

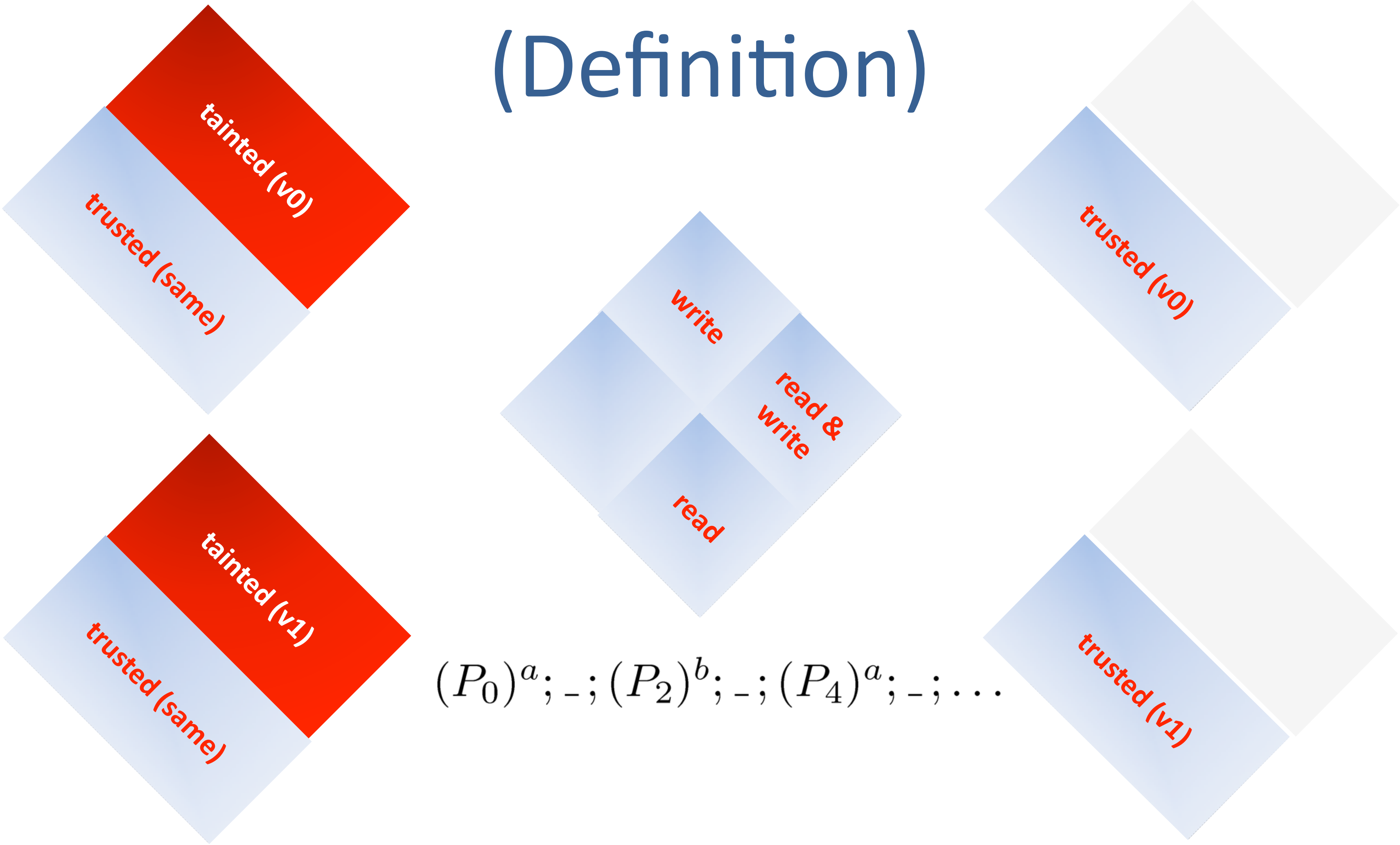
(Definition)



$$(P_0)^a;\_;(P_2)^b;\_;(P_3)^a;\_;(P_4)^b;\ \ldots$$

1. **Pick two initial memories with same trusted values**
2. **Run (twice) the same program interleaved with any adversary code**
3. **Do we obtain the same trusted results?**

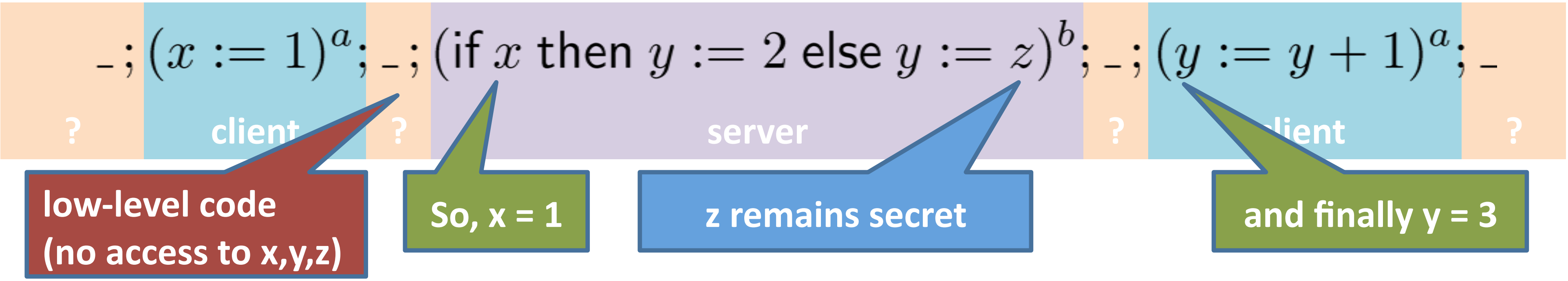
Integrity (Definition)



$$(P_0)^a; \_; (P_2)^b; \_; (P_4)^a; \_; \ldots$$

1. **Pick two initial memories with the same trusted values**
2. **Run (twice) the same program interleaved with any adversary code**
3. **Do we obtain the same trusted results?**

Sample Source Code



$$\_;(x := 1)^a;\_;(\text{if } x \text{ then } y := 2 \text{ else } y := z)^b;\_;(y := y + 1)^a;\_$$

low-level code
(no access to x,y,z)

So, x = 1

z remains secret

and finally y = 3

- High-level variables are protected by the memory policy ¡

- In a less abstract implementation,
  host *a* must pass *x* securely to host *b*; then
  host *b* must pass *y* securely to host a; ...

Sample Implementation

We implement $(\text{if } x \text{ then } y := 2 \text{ else } y := z)^b$

server

***x* is shared via low level variables x_e x_m**

x_e contains *x*'s encrypted value

x_m contains x_e's crypto MAC

To read x, we verify the MAC...

... then we decrypt x_e into a **local** secure variable x^b

$$
\begin{aligned}
&\text{if } \mathbf{Verify}(x_e, x_m, k_m) \text{ then } (\\
&\quad x^b := \mathbf{Decrypt}(x_e, k_e); \\
&\quad \text{if } x^b \text{ then } y^b := 2 \text{ else } y^b := z^b; \\
&\quad y_e := \mathbf{Encrypt}(y^b, k_e); y_m := \mathbf{MAC}(y_e, k_m))
\end{aligned}
$$

- Which crypto primitives? Which keys?
  Does it provide the same security?
- When to run this code?

Sample Implementation

We implement $(\text{if } x \text{ then } y := 2 \text{ else } y := z)^b$

server

***x* is shared via low level variables x_e x_m**

x_e contains *x*'s encrypted value

x_m contains x_e's crypto MAC

To read x, we verify the MAC...

... then we decrypt x_e into a **local** secure variable x^b

$$
\begin{aligned}
&\text{if } \mathbf{Verify}(x_e, x_m, k_m) \text{ then } (\\
&\quad x^b := \mathbf{Decrypt}(x_e, k_e); \\
&\quad \text{if } x^b \text{ then } y^b := 2 \text{ else } y^b := z^b; \\
&\quad y_e := \mathbf{Encrypt}(y^b, k_e); y_m := \mathbf{MAC}(y_e, k_m))
\end{aligned}
$$

- Here, we cannot use the same key for MACing *x* and *y*
  - the adversary code $y_e := x_e; y_s := x_s$ can achieve $y := x$.

Sample Implementation

We implement $(\text{if } x \text{ then } y := 2 \text{ else } y := z)^b$

server

x is shared via low level variables x_e x_m

x_e contains x's encrypted value

x_m contains x_e's crypto MAC

To read x, we verify the MAC...

... then we decrypt x_e into a **local** secure variable x^b

$$
\begin{aligned}
&\text{if } \mathbf{Verify}(x_e, x_m, k_m) \text{ then } (\\
&\quad x^b := \mathbf{Decrypt}(x_e, k_d); \\
&\quad \text{if } x^b \text{ then } y^b := 2 \text{ else } y^b := z^b; \\
&\quad y_e := \mathbf{Encrypt}(y^b, k_e); y_m := \mathbf{MAC}(y_e, k_m))
\end{aligned}
$$

- Here, we cannot rely on the same key for protecting *x* and *y*
  - If we insert the code $y_e := x_e; y_s := x_s$ between b and a, we achieve $y = 2$
- Besides, the adversary can "break" integrity using $x_s := 0$

Accommodating Runtime Errors

- Integrity non-interference (rightfully) excludes implicit flows

$$P[\_, Q] \doteq l := 4; \_; \mathsf{if}\ l = 4\ \mathsf{then}\ h := 10\ \mathsf{else}\ Q$$

After running $P[\mathsf{skip}, h := 5]$ we have $h = 10$
After running $P[l := 0, h := 5]$, we have instead $h = 5$ (implicit flow from l to h)

- Any dynamic checks create “implicit” flows!
  - E.g. we dynamically check whether a signature is correct

- We refine our model to accommodate runtime errors
  - ***If the program completes***, then it guarantees integrity
  - The command context $P[\_, \mathsf{skip}]$ is well-typed, as it preserves the integrity of *h* (or leaves *h* uninitialized)

INFORMATION-FLOW SECURITY (REVIEW)

CRYPTOGRAPHIC PROTECTION FOR SHARED MEMORY

A LANGUAGE FOR COMPUTATIONAL CRYPTOGRAPHY

OUR PROTOTYPE COMPILER

Cryptographic Assumptions: CPA

An encryption scheme is any triple $(\mathbf{KeyGen}, \mathbf{Encrypt}, \mathbf{Decrypt})$
of probabilistic polytime functions with two properties

Correctness:

$$\mathbf{Decrypt}(\mathbf{Encrypt}(x, k_e); k_d) = x$$

Security e.g. against chosen-plaintext attacks (CPA):
a probabilistic polytime game

- The adversary (_) passes any pair of values to an encryption oracle
  The encryption oracle (*E)* encrypts either the first value, or the second value
- The adversary knows this ciphertext and the encryption key
- The adversary wins if it guesses which value is encrypting

In our language:

$$\begin{aligned} CPA \doteq\ & b := \{0,1\}; \\ & k_e, k_d := \mathbf{KeyGen}; \\ & \_; E; \_ \end{aligned}$$

$$\begin{aligned} E \doteq\ & \text{if } b \\ & \quad \text{then } x := x_0 \\ & \quad \text{else } x := x_1; \\ & x_e := \mathbf{Encrypt}(x, k_e) \end{aligned}$$

$$Pr(CPA; b = g) \leq \frac{1}{2} + \epsilon(\eta)$$

if b
then $x := x_0$
else $x := x_1$

$b := \{0, 1\}$

$x_0 := 5$
$x_1 := 12$
$x_e := \mathbf{Encrypt}(x, k_e)$
$\dots$
$g := 1$

$k_e := \mathbf{KeyGen}()$

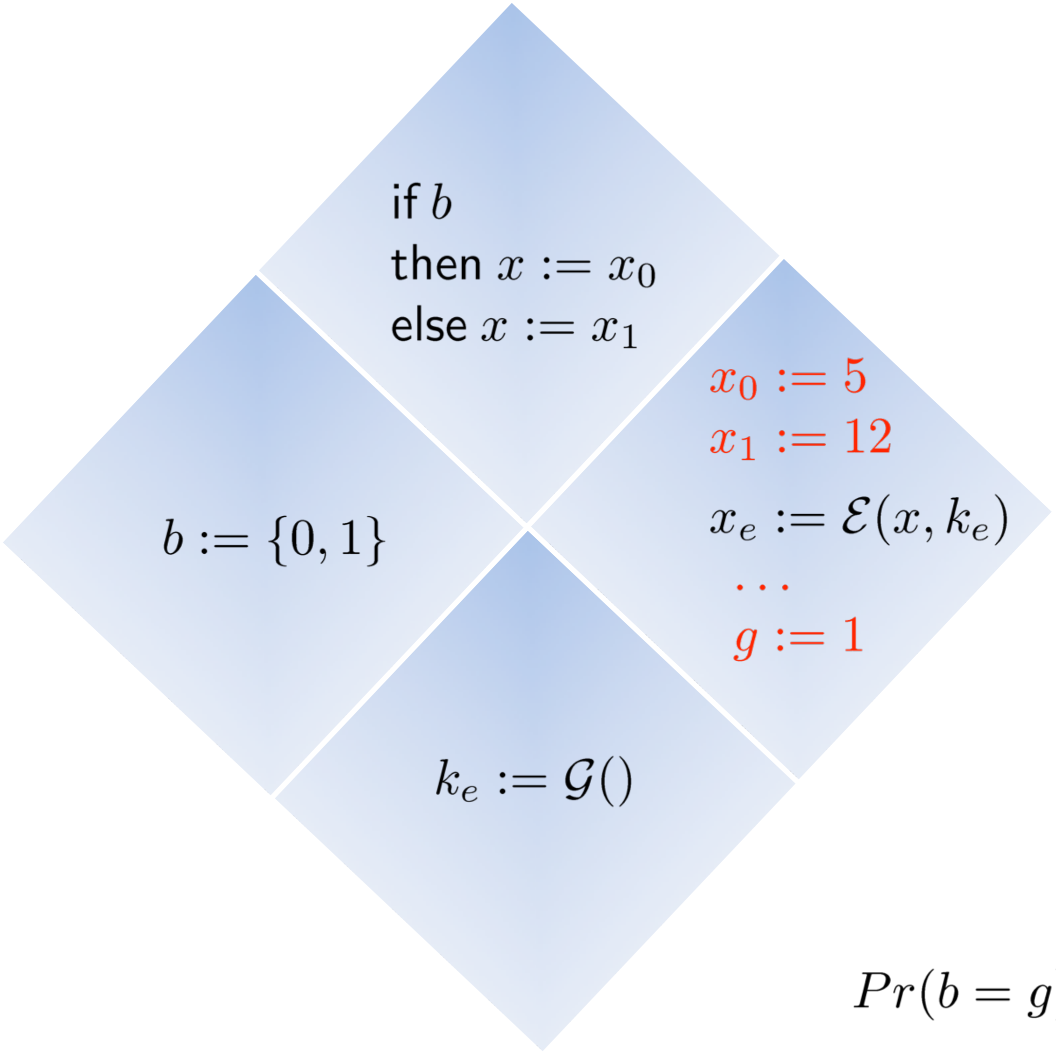
In our language:

$$
\begin{aligned}
CPA \doteq\; & b := \{0, 1\}; \\
& k_e, k_d := \mathbf{KeyGen}; \\
& \_; E; \_
\end{aligned}
$$

$$Pr(CPA; b = g) \leq \frac{1}{2} + \epsilon(\eta)$$

$$
\begin{aligned}
E \doteq\; & \text{if } b \\
& \quad \text{then } x := x_0 \\
& \quad \text{else } x := x_1; \\
& x_e := \mathbf{Encrypt}(x, k_e)
\end{aligned}
$$

Secrecy for a Single Encryption



$b := \{0,1\}$

$k_e := \mathcal{G}()$

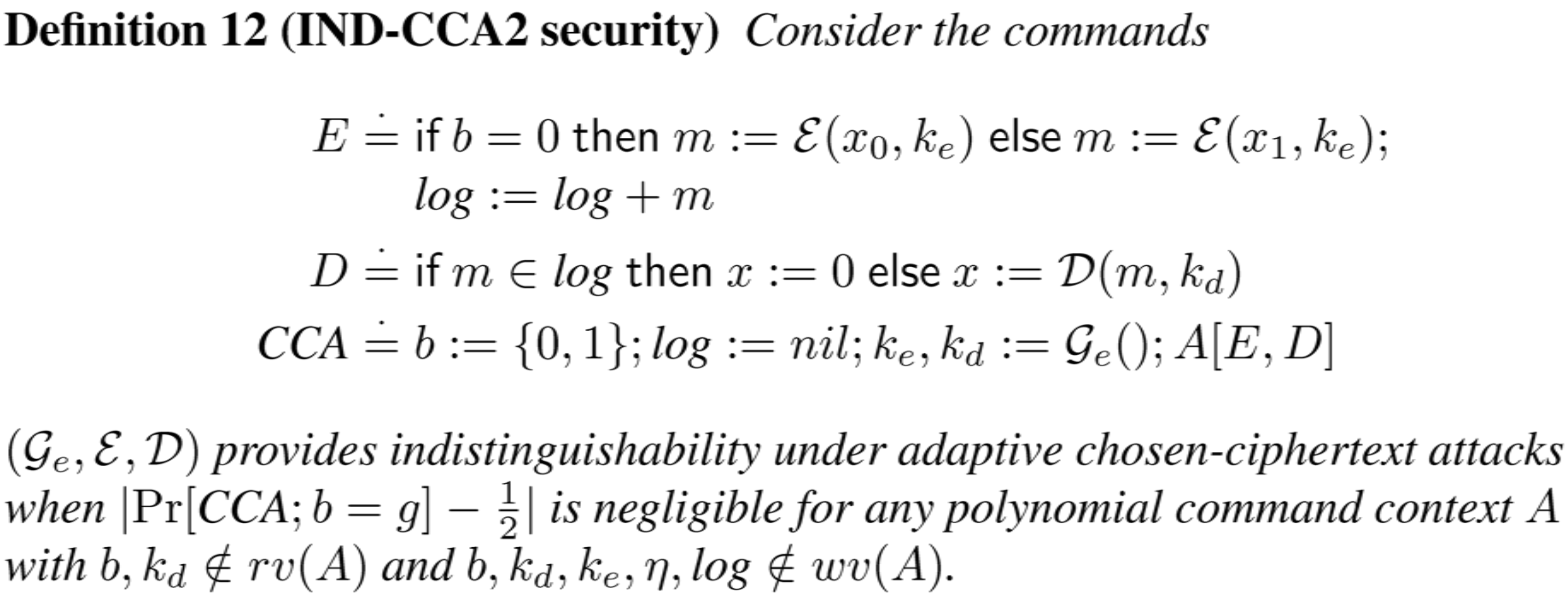
if b
then $x := x_0$
else $x := x_1$

$x_0 := 5$
$x_1 := 12$
$x_e := \mathcal{E}(x, k_e)$
$\dots$
$g := 1$

$$Pr(b = g) \leq \frac{1}{2} + \epsilon(\eta)$$

Cryptographic Assumptions (Sample)

- An encryption scheme is a triple $(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ of probabilistic polynomial functions expressible in the target language.
- We rely on a (command-based) standard security assumption:



Definition 12 (IND-CCA2 security) *Consider the commands*

$$\begin{aligned} E &\doteq \text{if } b = 0 \text{ then } m := \mathcal{E}(x_0, k_e) \text{ else } m := \mathcal{E}(x_1, k_e); \\ &\quad\; \mathit{log} := \mathit{log} + m \\ D &\doteq \text{if } m \in \mathit{log} \text{ then } x := 0 \text{ else } x := \mathcal{D}(m, k_d) \\ CCA &\doteq b := \{0,1\}; \mathit{log} := \mathit{nil}; k_e, k_d := \mathcal{G}_e(); A[E, D] \end{aligned}$$

$(\mathcal{G}_e, \mathcal{E}, \mathcal{D})$ *provides indistinguishability under adaptive chosen-ciphertext attacks when* $|\Pr[CCA; b = g] - \frac{1}{2}|$ *is negligible for any polynomial command context* A *with* $b, k_d \notin rv(A)$ *and* $b, k_d, k_e, \eta, \mathit{log} \notin wv(A)$.

- We similarly rely on integrity properties for signatures and MACS

INFORMATION-FLOW SECURITY (REVIEW)
CRYPTOGRAPHIC PROTECTION FOR SHARED MEMORY
MODELLING COMPUTATIONAL CRYPTOGRAPHY
PROTOTYPE COMPILER

Source Language: Adding Locality

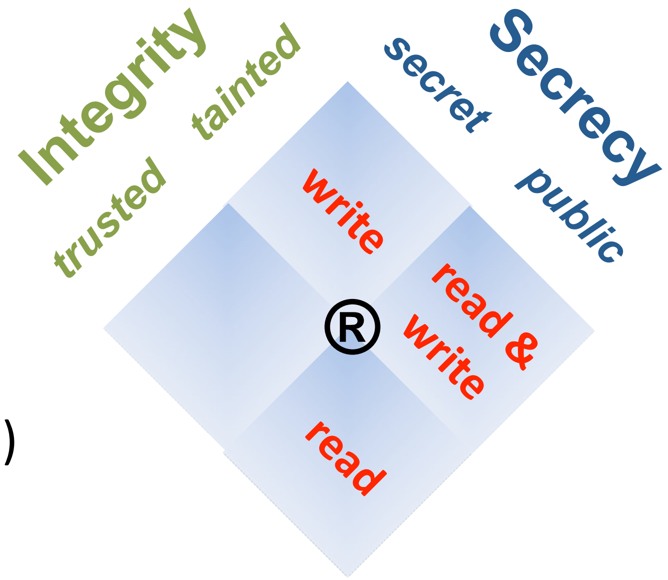
- Source commands are annotated with **locations** representing principals, machines ,etc

$$P \;::= (P)^a \mid \ldots$$

- A **security policy ¡** maps every variable to an information level
- A program consists of code fragments
  - Running on different hosts
  - Sharing the memory

$$(P_0)^a \;;\; (P_1)^c \;;\; (P_2; (P_3)^a; P_4)^b; \ldots$$

Active Adversaries



- An adversary is specified as a **compromise level**
  - Can read/write shared memory
  - Can control code at lower-level hosts (e.g. c)

- After specifying the adversary, we erase lower level code

$$(P_0)^a \; ; \; (P_1)^c \; ; \; \left(P_2;(P_3)^a;P_4\right)^b ; \ldots$$

$$\rightsquigarrow \; (P_0)^a;\_;(P_2)^b;\_;(P_3)^a;\_;(P_4)^b; \; \ldots$$

("_" stands for any adversary code)

The CFLOW Compiler

- It takes a source program with locality annotations

$$P ::= (P)^{a}\ j\ :::$$

… and yield a series of local commands

$$C(P) = Q_0; Q_a; Q_b; :::$$

1. Compiled code behave as the source ***when fairly scheduled***
3. Compiled code is as secure as the source ***when controlled by the adversary***

$$Q_0; A[Q_a; Q_b; :::]$$

The Compiler Extension

- It takes a series of local commands with
  - One command marked to be implemented with hardware capabilities
  - A set of variables local to this command to protect

$$\tilde{Q} = Q_0; Q_a; Q_b; :::; Q_v$$

… and yield a new series of commands where Q_v ask for less privileges

$$C(\tilde{Q}) = Q^0_0; Q^0_a; Q^0_b; :::; Q^0_v$$

Compiler (Definition)

- A compiler takes
  a source program with
  locality annotations

$$P \;::=\; (P)^a \mid \ldots$$

  … and yield a series
  of local commands

$$\mathcal{C}(P) = Q_0, Q_a, Q_b, \ldots$$

- These commands can
  be explicitly scheduled,
  e.g. in a round-robin

$$\begin{aligned} N[Q_a, Q_b] \;&= \\ &\quad \mathit{next} := \mathit{start}; \\ &\quad \text{while } \mathit{next} \neq \mathit{stop} \text{ do } \{Q_a; Q_b\} \end{aligned}$$

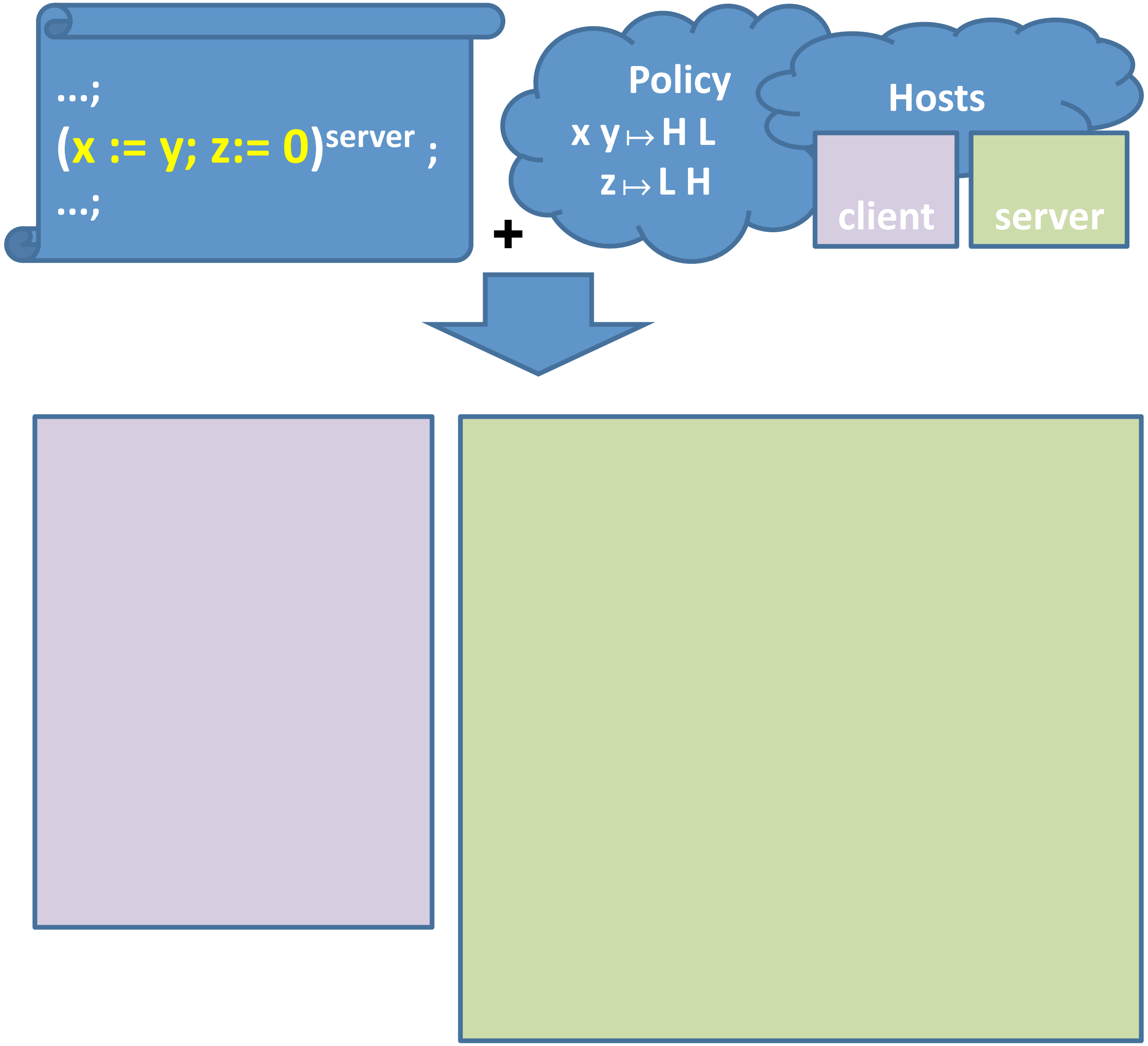
Compiler (Theorems)

1. Compiled code $\mathcal{C}(P) = Q_0, Q_a, Q_b$
   behaves as the source ***when fairly scheduled***:
   - For all initial memories,
     final memories after running P have the same distribution as
     final memories after running $Q_0; N[Q_a, Q_b]$

3. Compiled code is as secure as the source
   when controlled by the adversary
   - If, for any two initial memories,
     $\widehat{P}$ preserves confidentiality for all ® adversaries, then also
     $Q_0; \_[Q_a, Q_b]$ preserves confidentiality for all ® adversaries

   (and similarly for integrity)

Cryptographic Compilation



…;

$(x := y;\ z := 0)^{server}$;

…;

+

Policy

x y ↦ H L

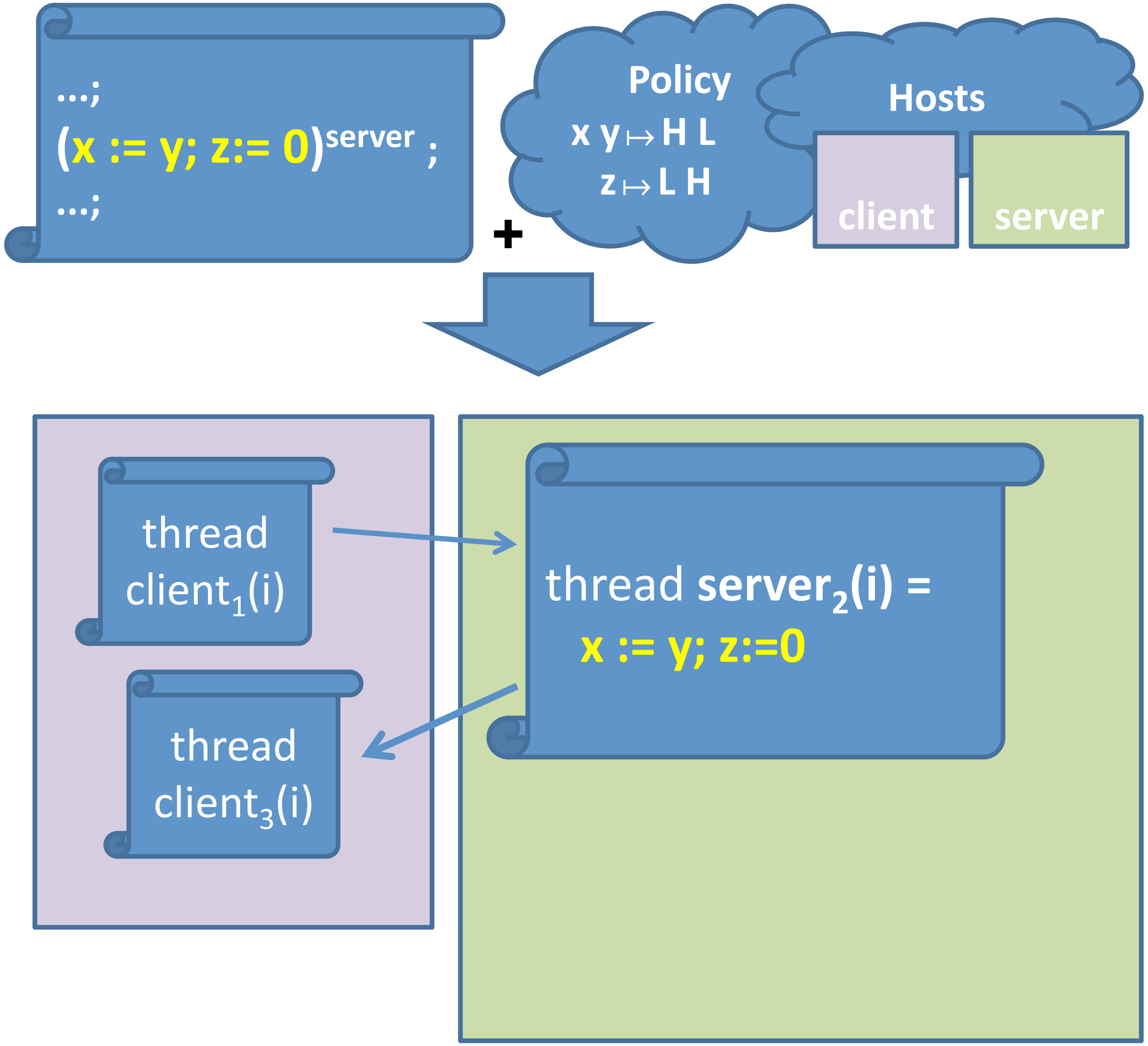
z ↦ L H

Hosts

client

server

Cryptographic Compilation



…;

(x := y; z:= 0)$^{\text{server}}$;

…;

+

Policy

x y ↦ H L

z ↦ L H

Hosts

client

server

thread client$_1$(i)

thread **server$_2$(i) =**

x := y; z:=0

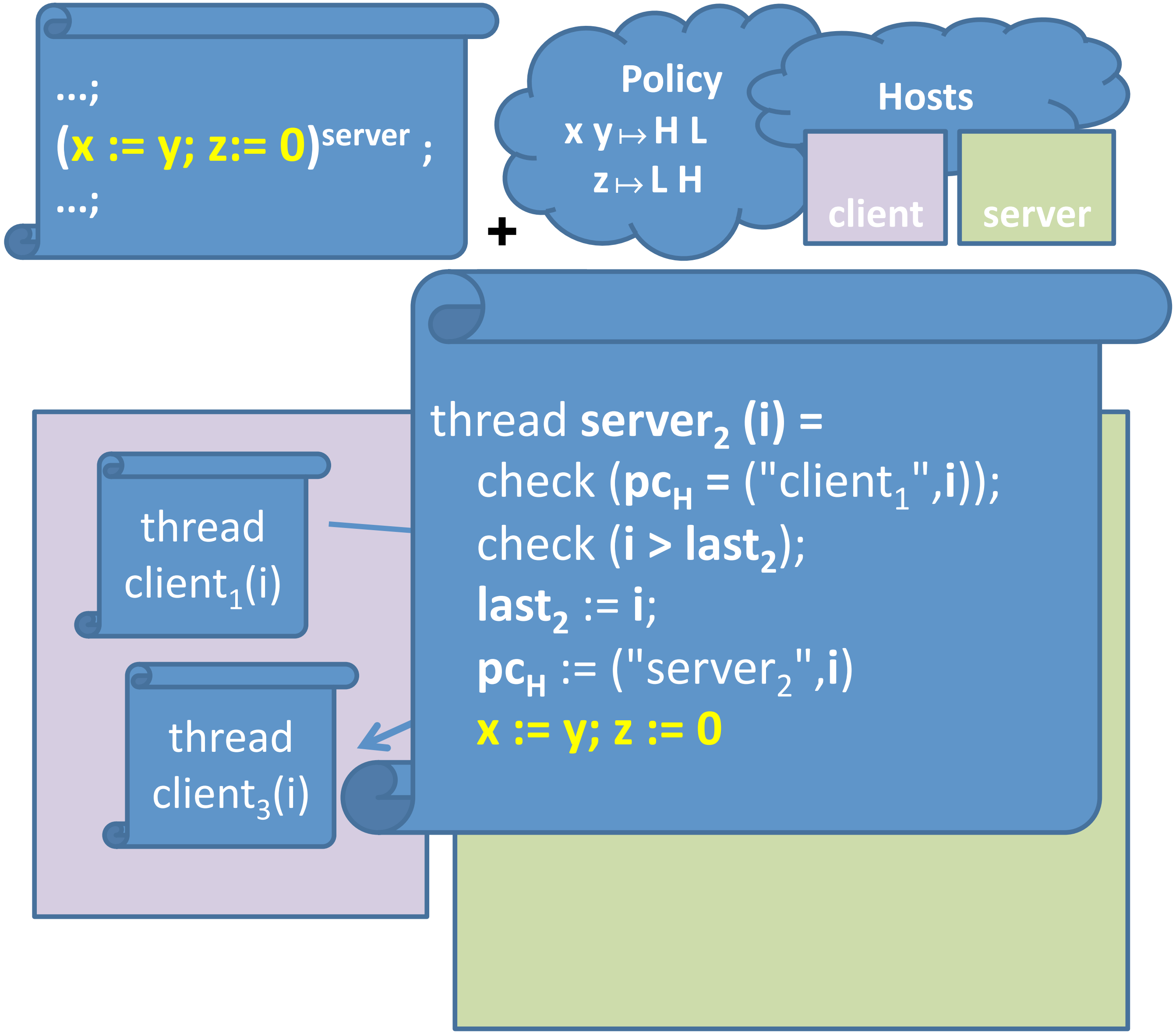
thread client$_3$(i)

1. **Split the program into local threads.**

Each thread:

- has a fixed integrity level
- is parameterized by **loop indexes** (+1 for each loop)
- runs **just once** at every index (for anti-replay)
- is called by **at most one** remote thread (for integrity enforcement)

Cryptographic Compilation



```
…;
(x := y; z:= 0)$^{server}$ ;
…;
```

+

```
thread server$_2$ (i) =
    check (pc$_H$ = ("client$_1$",i));
    check (i > last$_2$);
    last$_2$ := i;
    pc$_H$ := ("server$_2$",i)
    x := y; z := 0
```

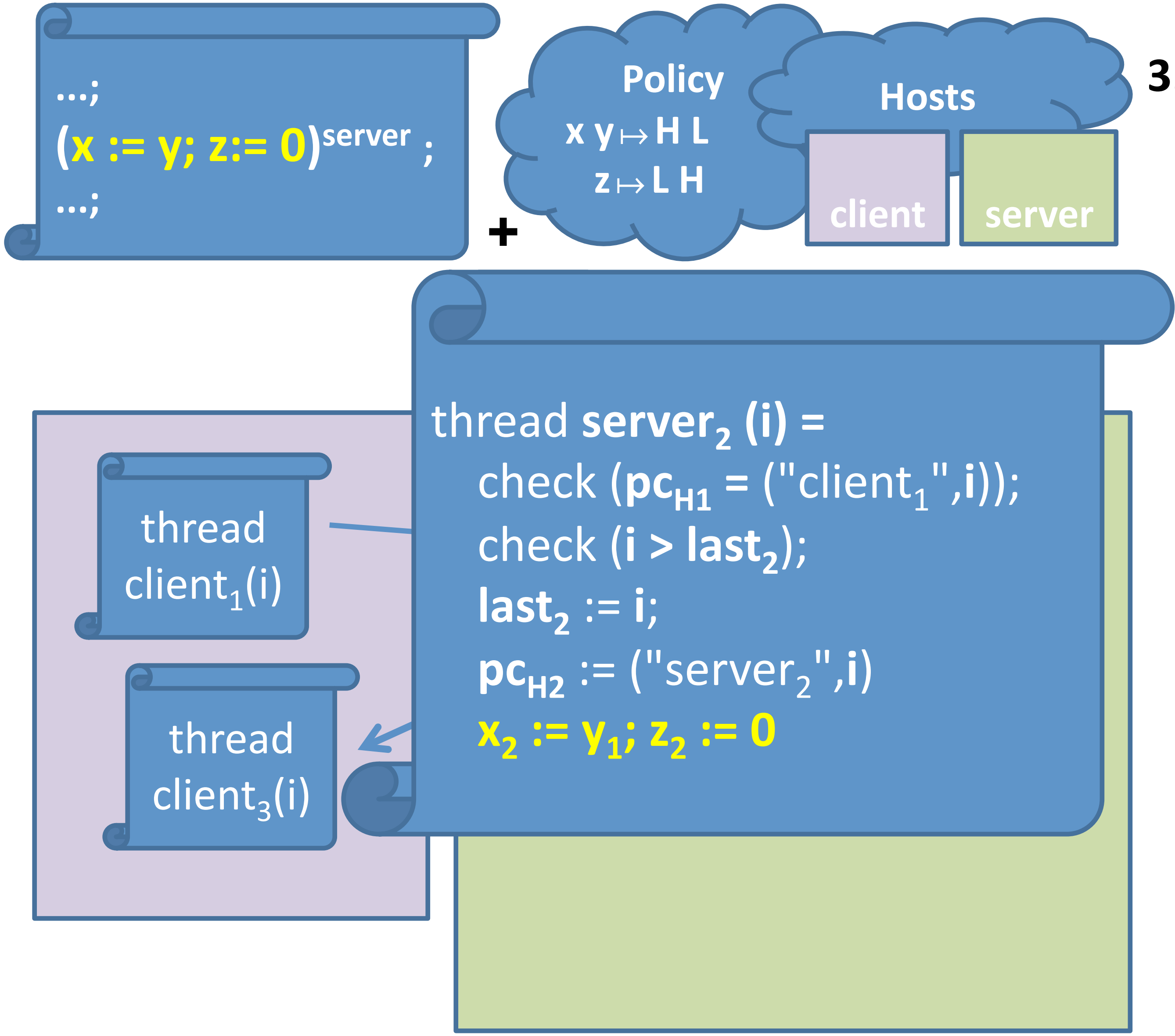
1. Split the program into local threads
2. **Secure control flow using program counters**

(one shared PC at each integrity level)

Before running a thread:

- check that PCs have their expected values
- test & increment local anti-replay counter
- update PC at the thread integrity

Cryptographic Compilation



…;
(x := y; z:= 0)$^{\text{server}}$;
…;

\+

Policy
x y ↦ H L
z ↦ L H

Hosts
client
server

thread
client$_1$(i)

thread
client$_3$(i)

thread **server$_2$ (i)** =
check (**pc$_{H1}$** = ("client$_1$",**i**));
check (**i** > **last$_2$**);
last$_2$:= **i**;
pc$_{H2}$:= ("server$_2$",**i**)
x_2 := y_1; z_2 := 0

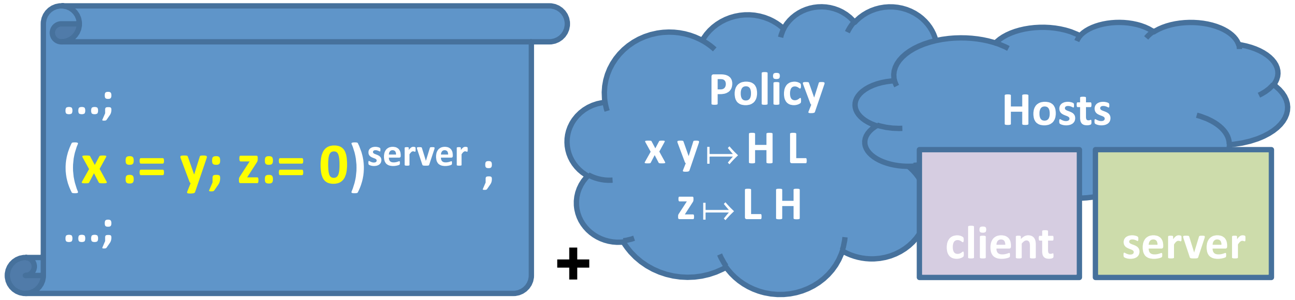
1. Split the program into local threads
2. Secure control flow, using program counters
3. **Split shared variables into local replicas** (single, static assign)

We use a variant of SSA to track writers (fixpoint computation)

We allocate a replica for each thread that accesses the variable

We explicitly propagate updates between hosts

Cryptographic Compilation



…;
$(x := y;\ z := 0)^{server}$;
…;

\+

Policy
x y ↦ H L
z ↦ L H

Hosts
client
server

```
thread $server_2$ (i) =
    Verify ("$client_1$." + i + ".$pc_H$." + $pc_{H1}$)  $pc_{Hm}$ $k_{m1}$;
    $y_2$ := Decrypt ($y_e$ ,$k_{e1}$);
    check ($pc_{H1}$ = ("$client_1$",i));
    check (i > $last_2$);
    $last_2$ := i;
    $pc_{H2}$ := ("$server_2$",i)
    $x_2$ := $y_1$; $z_2$ := 0;
    $pc_{Hm}$  := MAC ("$server_2$." + i + ".$pc_H$." + $pc_{H2}$) $k_{m1}$;
```

1. Split the program into local threads
2. Secure control flow, using program counters
3. Split shared variables into local replicas
4. **Cryptographically protect reads and writes**

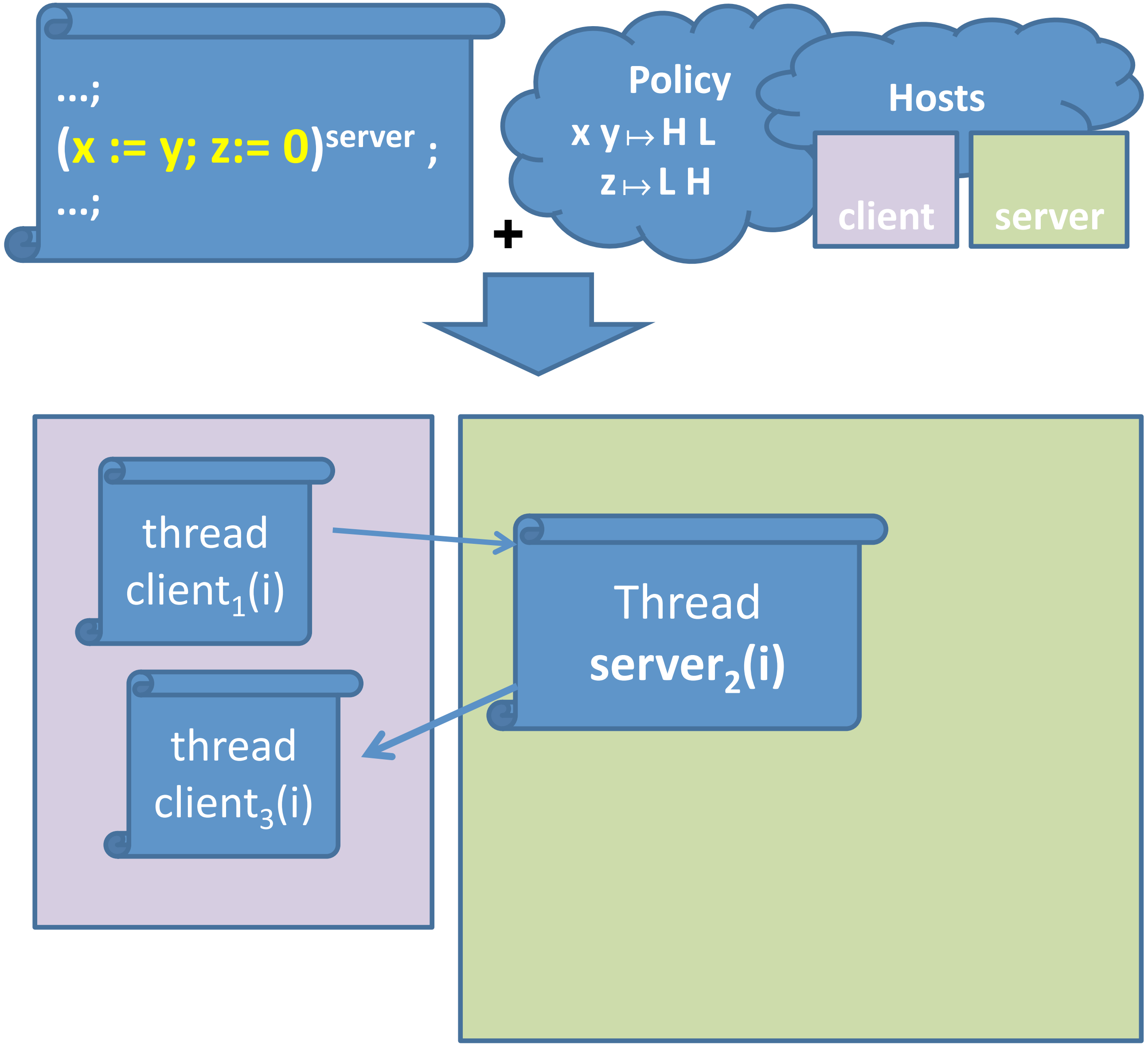
   encrypt and/or sign at each remote call

   use auxiliary keys (greedy allocation)

   use long-lived PKI only for bootstrapping

All replicas shared between threads have low integrity and confidentiality

Cryptographic Compilation



…;
(x := y; z:= 0)^{server} ;
…;

+

Policy
x y ↦ H L
z ↦ L H

Hosts
client
server

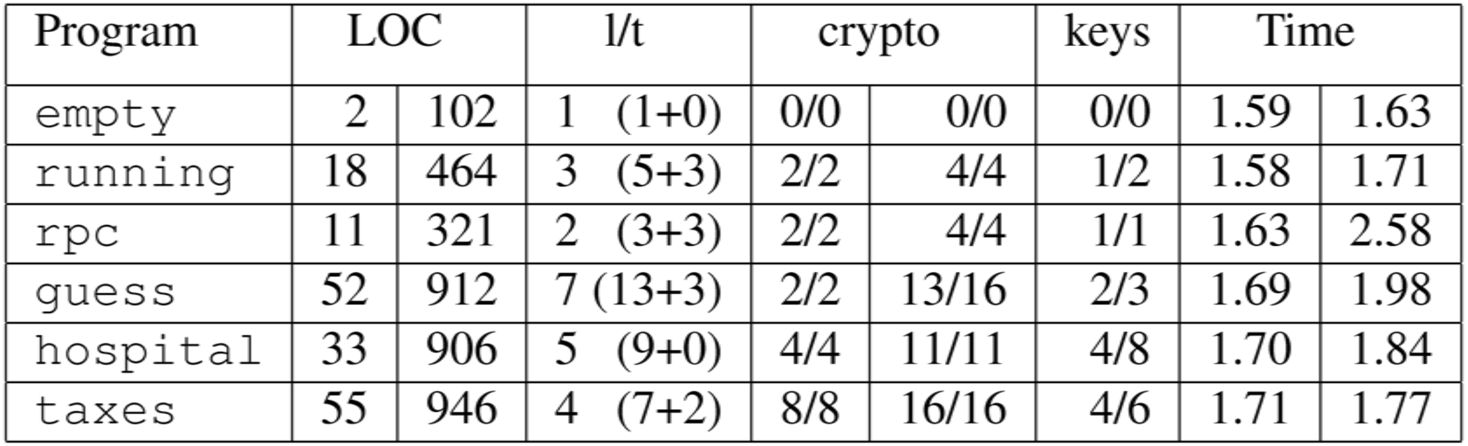
thread client$_1$(i)

Thread **server$_2$(i)**

thread client$_3$(i)

1. Split the program into local threads
2. Secure control flow, using program counters
3. Split shared variables into local replicas
4. Cryptographically protect reads and writes
5. **Generate untrusted code for scheduling and synchronization**

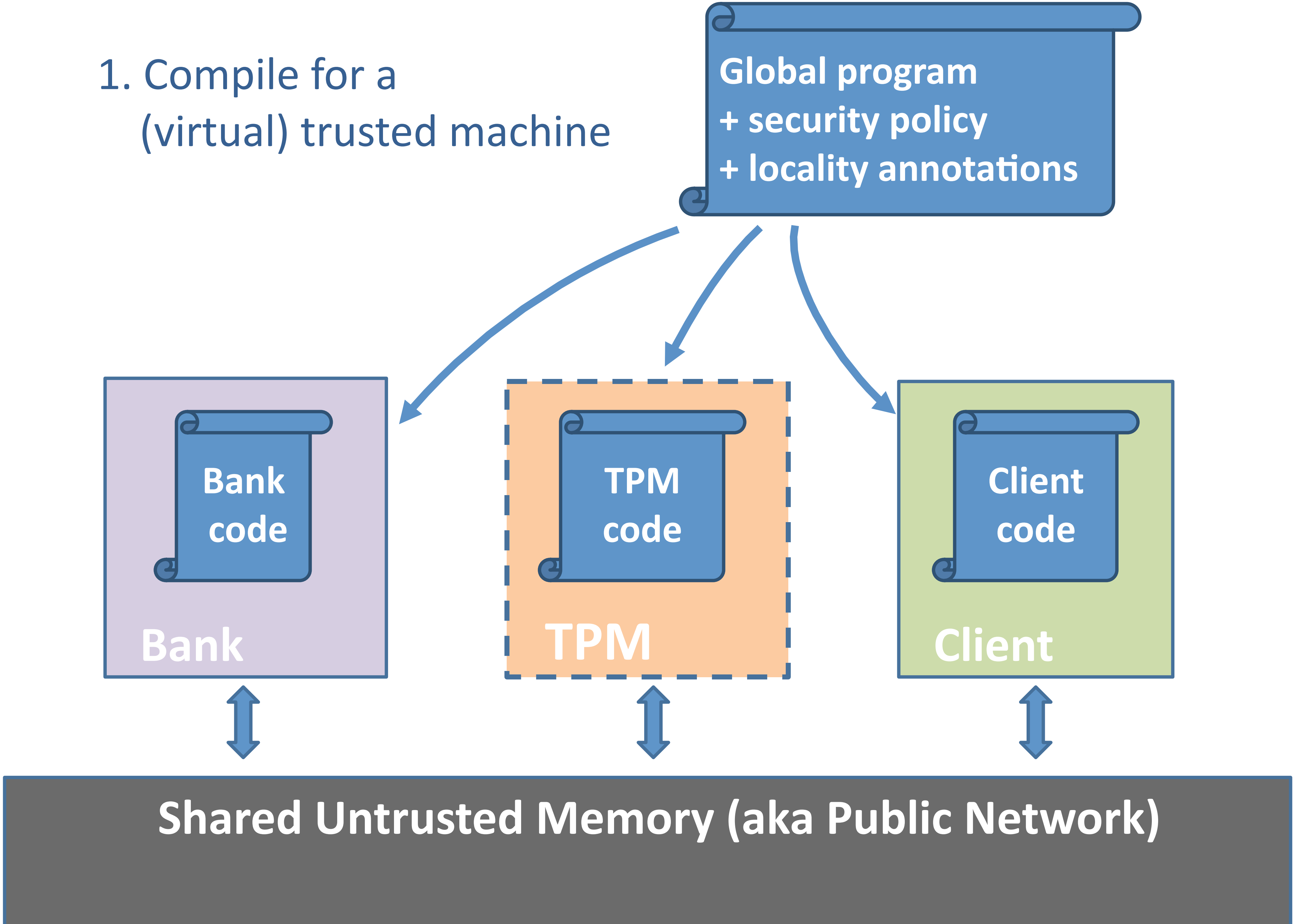
Experimental Results



Program	LOC		l/t	crypto		keys	Time	
`empty`	2	102	1 (1+0)	0/0	0/0	0/0	1.59	1.63
`running`	18	464	3 (5+3)	2/2	4/4	1/2	1.58	1.71
`rpc`	11	321	2 (3+3)	2/2	4/4	1/1	1.63	2.58
`guess`	52	912	7 (13+3)	2/2	13/16	2/3	1.69	1.98
`hospital`	33	906	5 (9+0)	4/4	11/11	4/8	1.70	1.84
`taxes`	55	946	4 (7+2)	8/8	16/16	4/6	1.71	1.77

- Our compiler is parameterized by a security lattice
  - we coded simple lattices and Myers' decentralized labels
- Source and target languages are subsets of F#
  - .NET libraries for communications and cryptography
  - Trusted configuration file for bootstrapping

1. Compile for a
(virtual) trusted machine



Global program
+ security policy
+ locality annotations

Bank code

Bank

TPM code

TPM

Client code

Client

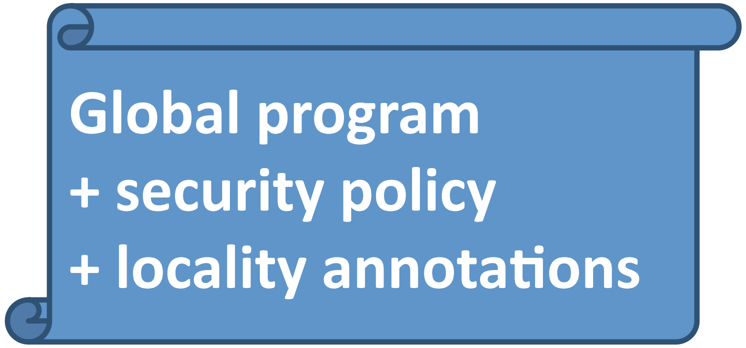
Shared Untrusted Memory (aka Public Network)

1. Compile for a (virtual) trusted machine

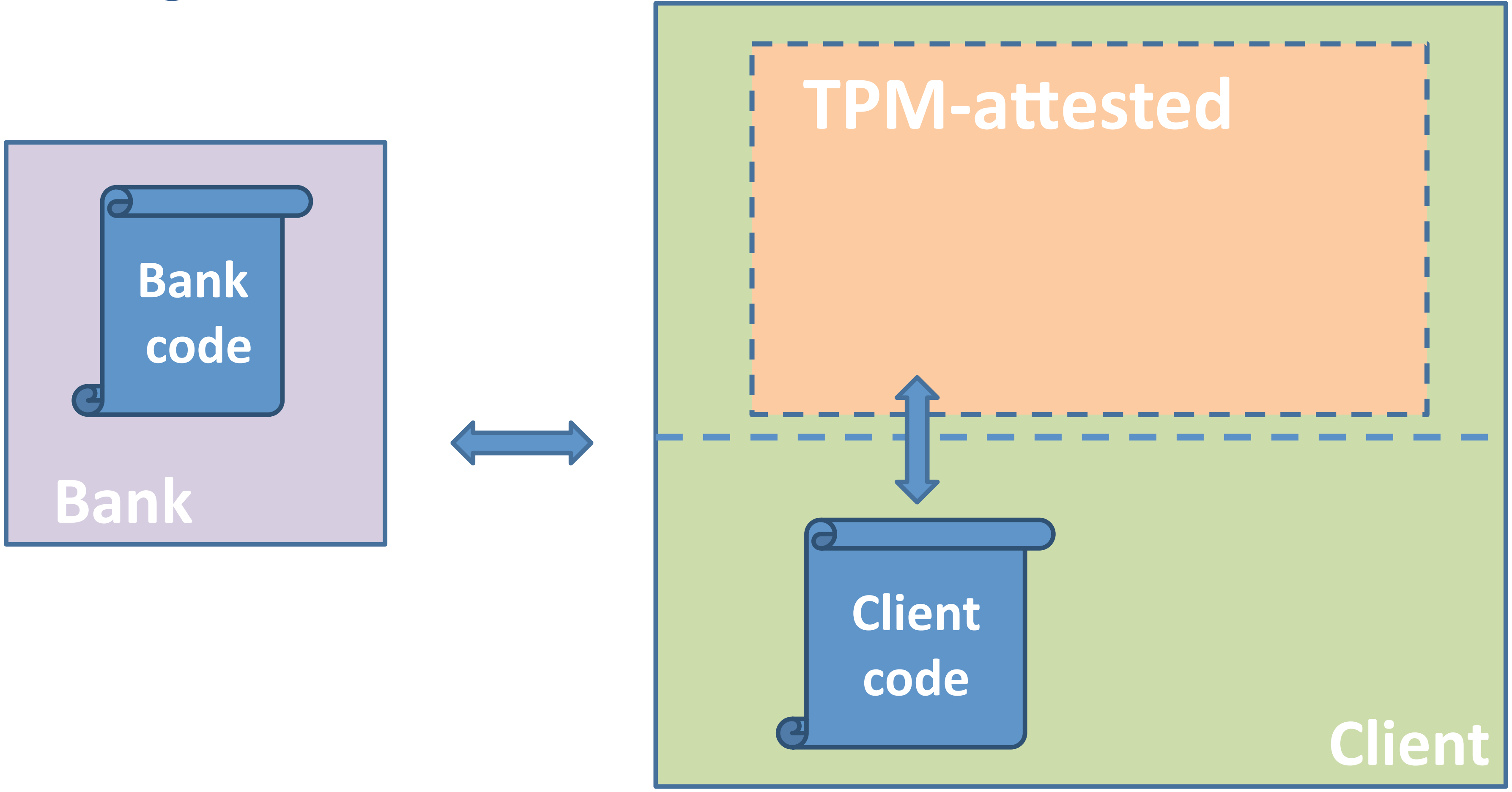
$$b\text{: }\{x_b := e_b\};\ c\text{:}\{y_c := e_c\};\ v\text{: }\{x'_b, y'_c := f(x_b, y_c)\};\ b\text{: }\{\mathit{print}(x'_b)\};\ c\text{: }\{\mathit{print}(y'_c)\}$$

$$\begin{aligned}
Q_0 &\doteq\quad k_b^-, k_b^+ := \mathcal{G}_e();\ k_v^-, k_v^+ := \mathcal{G}_e() \\
Q_b &\doteq\quad \text{if } c_b{=}1 \text{ then } \{\ c_b{+}{+};\ x_b := e_b;\ x_e := \mathcal{E}(x_b, k_v^+);\ x_s := \mathcal{S}(x_e, k_b^-)\ \} \\
&\qquad \text{else if } c_b = 2 \text{ then } \{\ c_b{+}{+};\ \text{if } \mathcal{V}(x'_e, x'_s, k_v^+) \text{ then } \mathit{print}(\mathcal{D}(x'_e, k_b^-))\ \} \\
Q_c &\doteq\quad \text{if } c_c{=}1 \text{ then } \{\ c_c{+}{+};\ y_c := e_c\ \} \text{ else if } c_c{=}2 \text{ then } \{\ c_c{+}{+};\ \mathit{print}(y'_c)\ \} \\
Q_v &\doteq\quad \text{if } c_v{=}1 \text{ then} \\
&\qquad \{\ c_v{+}{+};\ \text{if } \mathcal{V}(x_e, x_s, k_b^+) \text{ then } \{\ x_v := \mathcal{D}(x_e, k_v^-);\ x'_v, y'_c := f(x_v, y_c); \\
&\qquad\qquad x'_e := \mathcal{E}(x'_v, k_b^+);\ x'_s := \mathcal{S}(x_e, k_v^-)\ \}\ \}
\end{aligned}$$

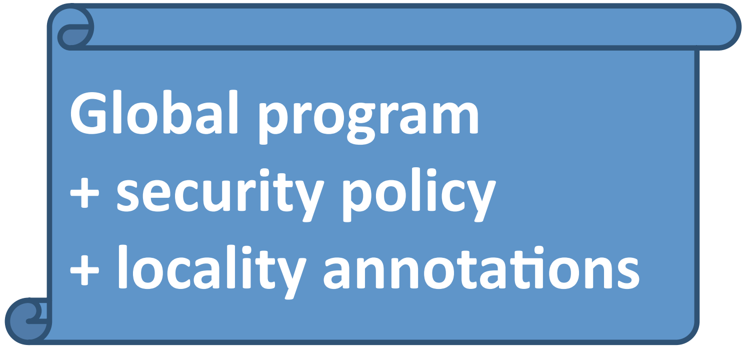
2. Generate code for dynamically booting and attesting the trusted machine using the client TPM



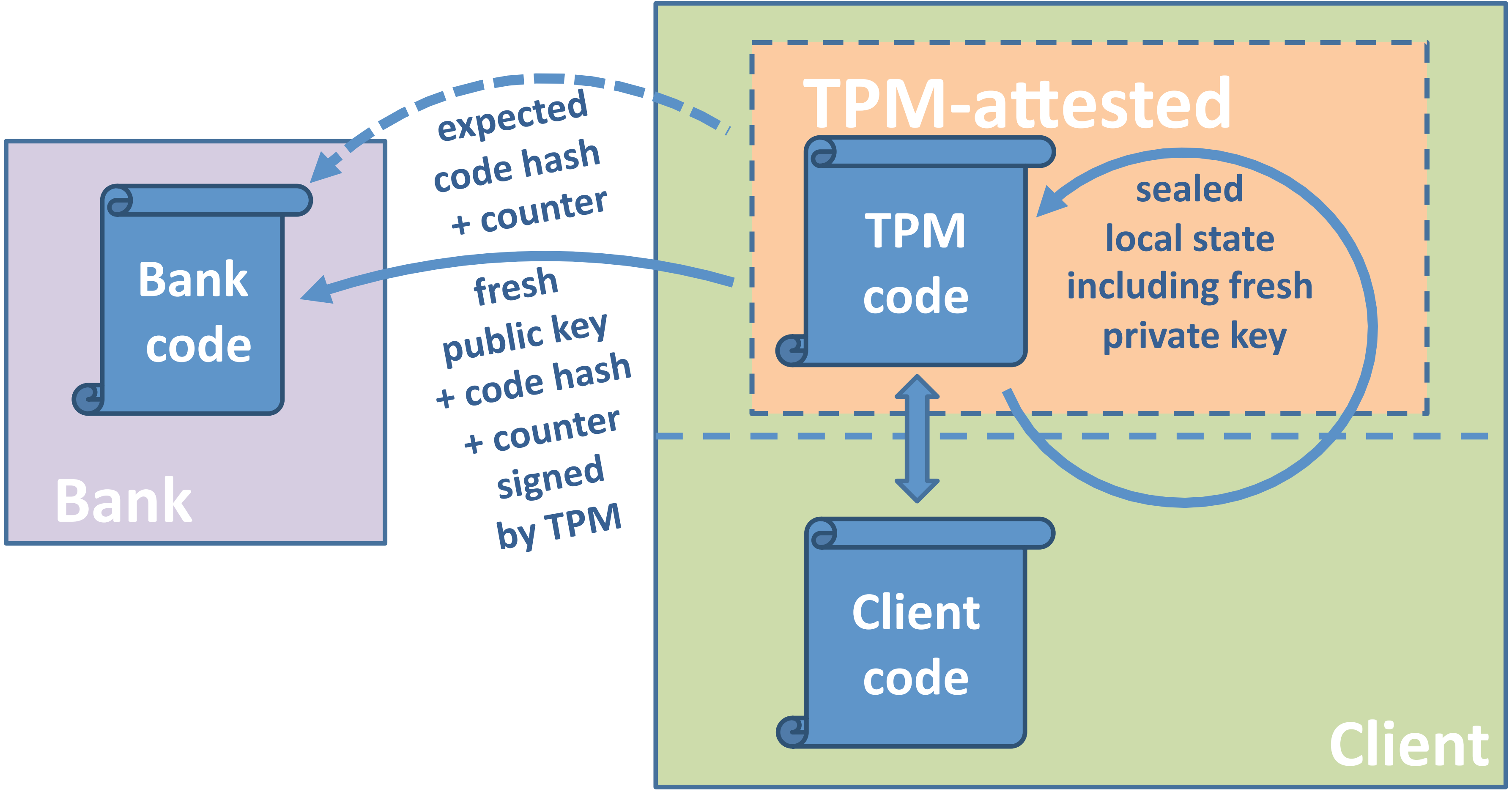
Global program
+ security policy
+ locality annotations



2. Generate code for dynamically booting and attesting the trusted machine using the client TPM



Global program
+ security policy
+ locality annotations



Bank code

Bank

expected
code hash
+ counter

fresh
public key
+ code hash
+ counter
signed
by TPM

TPM-attested

TPM code

sealed
local state
including fresh
private key

Client code

Client

2. Generate code for dynamically booting and attesting the trusted machine using the client TPM

Global program
+ security policy
+ locality annotations

```
$Q_0 \doteq$  $k_b^-, k_b^+ := \mathcal{G}_e();\ k_{TPM}^-, k_{TPM}^+ := \mathcal{G}_e();\ c := 0;$
$Q_b \doteq$  if $c_b$=1 then { $c_b$++; $x_b := e_b$;
                  if VERIFY($\mathcal{H}(\langle K_v \rangle), k_v^+, \mathit{cert}_v$)
                  [ $b.k_v^+ := k_v^+;\ x_e := \mathcal{E}(x_b, k_v^+);\ x_s := \mathcal{S}(x_e, k_b^-)$ ] }
      else if $c_b$=2 then { $c_b$++; if $\mathcal{V}(x'_e, x'_s, k_v^+)$ then $\mathit{print}(\mathcal{D}(x'_e, k_b^-))$ }
$Q_c \doteq$  if $c_c$=1 then { $c_c$++; $y_c := e_c$ } else if $c_c$=2 then { $\mathit{print}(y'_c)$ }
$Q_v \doteq$  $\mathit{kernel} := \langle K_v \rangle$; SKINIT
$K_v \doteq$  if $c$=0 then
        { INC; $k_v^-, k_v^+ := \mathcal{G}_e();\ \mathit{cert}_v :=$ ATTEST($k_v^+$); $\mathit{key} :=$ SEAL($k_v^-, h$) }
      else if $c$=1 then
        { INC; $k_v^- :=$ UNSEAL($\mathit{key}, h$);
          if $\mathcal{V}(x_e, x_s, k_b^+)$ then { $x_v := \mathcal{D}(x_e, k_v^-);\ x'_v, y'_c := f(x_v, y_c);$
                                 $x'_e := \mathcal{E}(x'_v, \text{‘}k_b^+);\ x'_s := \mathcal{S}(x'_e, k_v^-)$ } }
```

Theorems

- The compiler extension takes a series of local commands
  … and yields a new series of commands using the secure hardware primitives

$$\overset{e}{Q} = Q_0; Q_a; Q_b; : : : ; Q_v$$

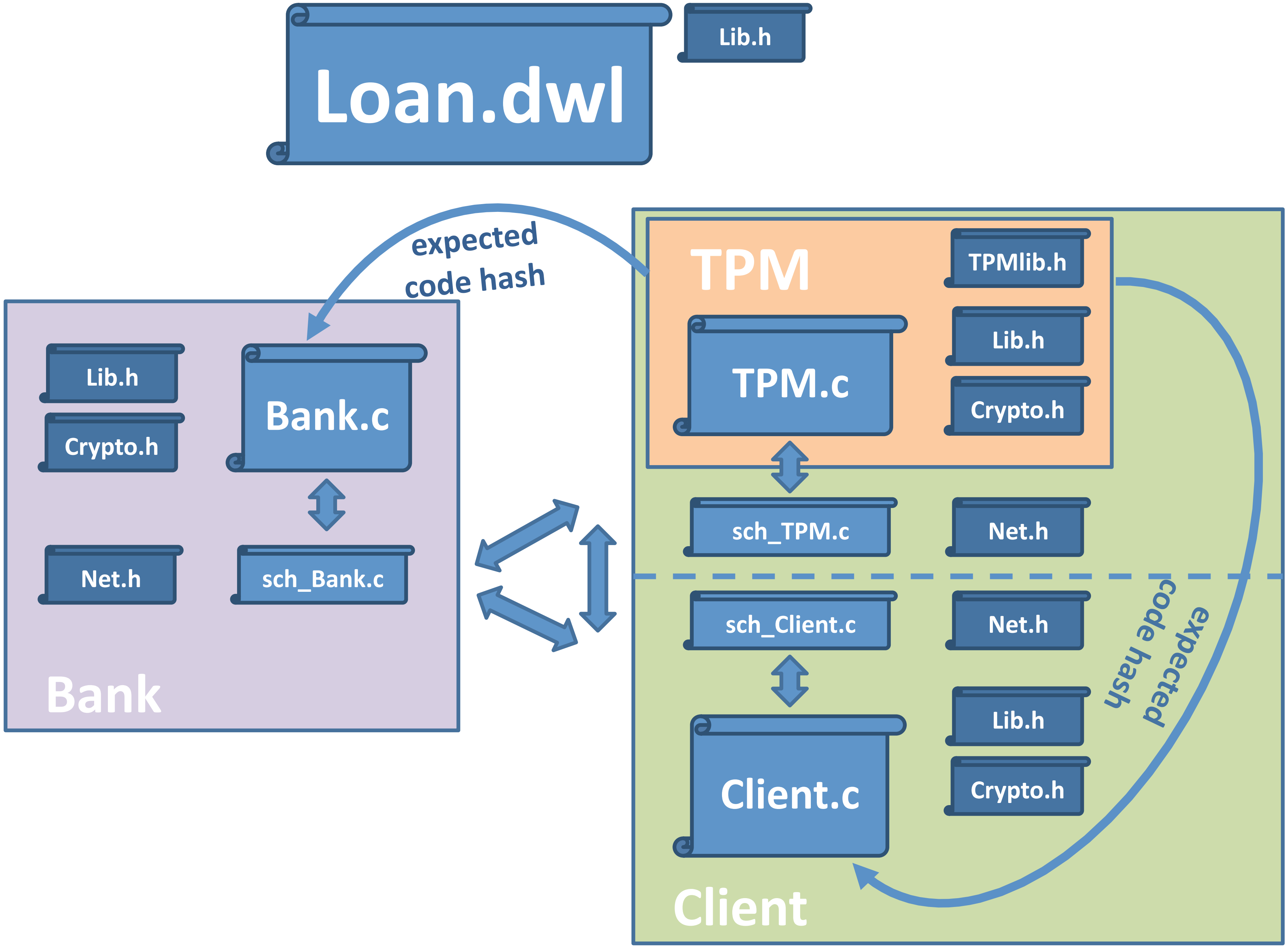
$$C(\overset{e}{Q}) = Q^0_0; Q^0_a; Q^0_b; : : : ; Q^0_v$$

- **Security**: For every adversary A’, there exists an A such that

$$Q_0; A[Q_a; Q_b; : : : ; Q_v] ¼ Q^{\upsilon}_0; A'[Q^{\upsilon}_a; Q^{\upsilon}_b; : : : ; Q^{\upsilon}_v]$$

- **Functionality**: For every scheduler A, there exists an A’ such that

$$Q_0; A[Q_a; Q_b; : : : ; Q_v] ¼ Q^{\upsilon}_0; A'[Q^{\upsilon}_a; Q^{\upsilon}_b; : : : ; Q^{\upsilon}_v]$$



Loan.dwl

Lib.h

expected code hash

Bank

Lib.h

Crypto.h

Bank.c

Net.h

sch_Bank.c

TPM

TPMlib.h

TPM.c

Lib.h

Crypto.h

sch_TPM.c

Net.h

sch_Client.c

Net.h

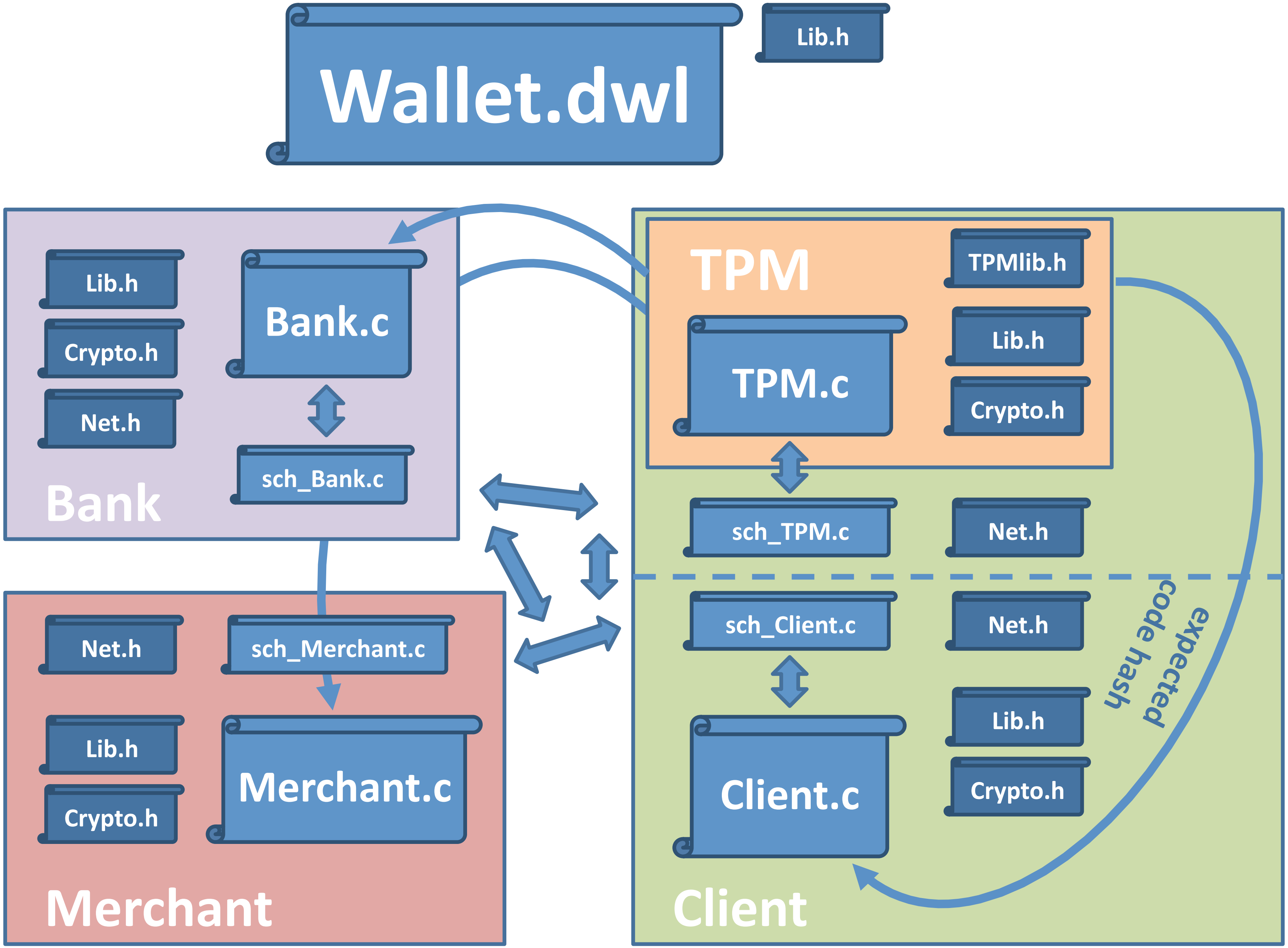
expected code hash

Client.c

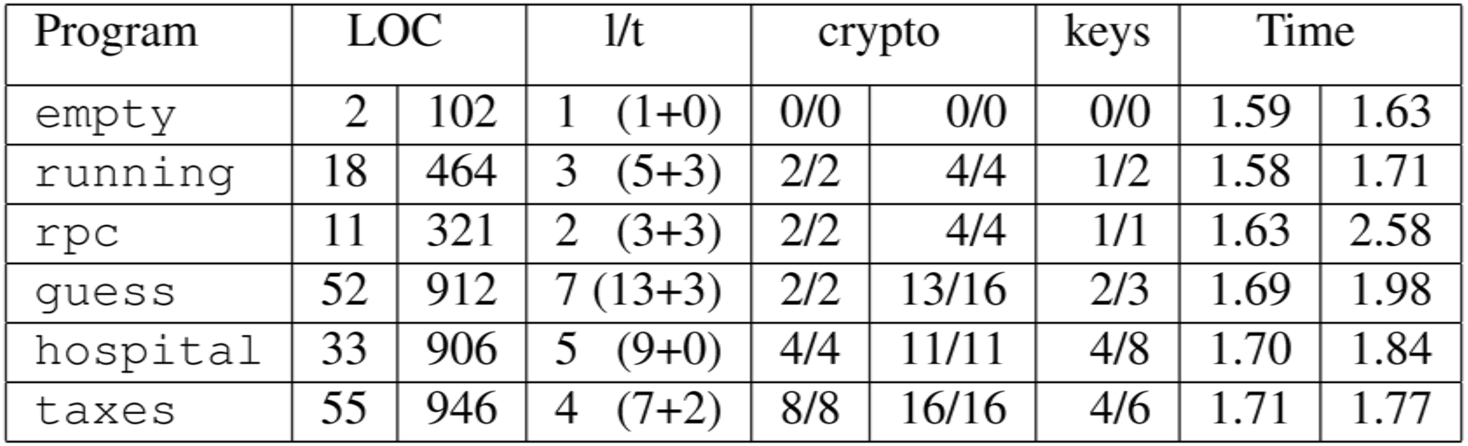
Lib.h

Crypto.h

Client


Wallet.dwl
Lib.h
Bank
Lib.h
Crypto.h
Net.h
Bank.c
sch_Bank.c
TPM
TPMlib.h
TPM.c
Lib.h
Crypto.h
sch_TPM.c
Net.h
Merchant
Net.h
sch_Merchant.c
Lib.h
Crypto.h
Merchant.c
Client
sch_Client.c
Net.h
Client.c
Lib.h
Crypto.h
expected code hash

Experimental Results



Program	LOC		l/t	crypto		keys	Time	
`empty`	2	102	1 (1+0)	0/0	0/0	0/0	1.59	1.63
`running`	18	464	3 (5+3)	2/2	4/4	1/2	1.58	1.71
`rpc`	11	321	2 (3+3)	2/2	4/4	1/1	1.63	2.58
`guess`	52	912	7 (13+3)	2/2	13/16	2/3	1.69	1.98
`hospital`	33	906	5 (9+0)	4/4	11/11	4/8	1.70	1.84
`taxes`	55	946	4 (7+2)	8/8	16/16	4/6	1.71	1.77

Summary

- We compile programs + security policies
  - With overwhelming probability,
    All information secrecy and integrity properties of the source program still hold in the implementation
- We account for active adversaries that control parts of the computation
- We target software/hardware minimal TCBs, relying on TPM-based secure instructions when available
- We rely on program transformations and standard (computational) cryptographic assumptions

http://www.msr-inria.inria.fr/projects/sec/cflow

Extra Slides

Related Work

- Non-interference
  Goguen & Meseguer 82, Bell & LaPadula 76, Denning 76
- Declassification
  Principles and dimensions: Sabelfeld & Sands 05
  Robust declassification: Zdancewic & Myers 01
  Enforcing robust declassification: Myers, Sabelfeld, Zdancewic 04
- Secure implementations
  Jif Split: Myers & Zheng 01
- Secure information flow and cryptography
  Laud 01—08, Backes & Pfitzmann 02—03
- Cryptographic DLM
  Vaughan & Zdancewic 07

Other Ongoing Projects

- We verify reference implementations
  for existing cryptographic protocols
  - Using **refinement types** [CSF'08] & cryptographic **model extraction** […]
  - Case study: TLS 1.0 [CCS'08[, CardSpace [ASIACCS'08]

- We generate cryptographic protocol code from security specs
  - *Shared-memory information flows* [POPL'08]
  - **Multiparty sessions** (communication graphs) [CSF'07]

Information-Flow Security (Review)

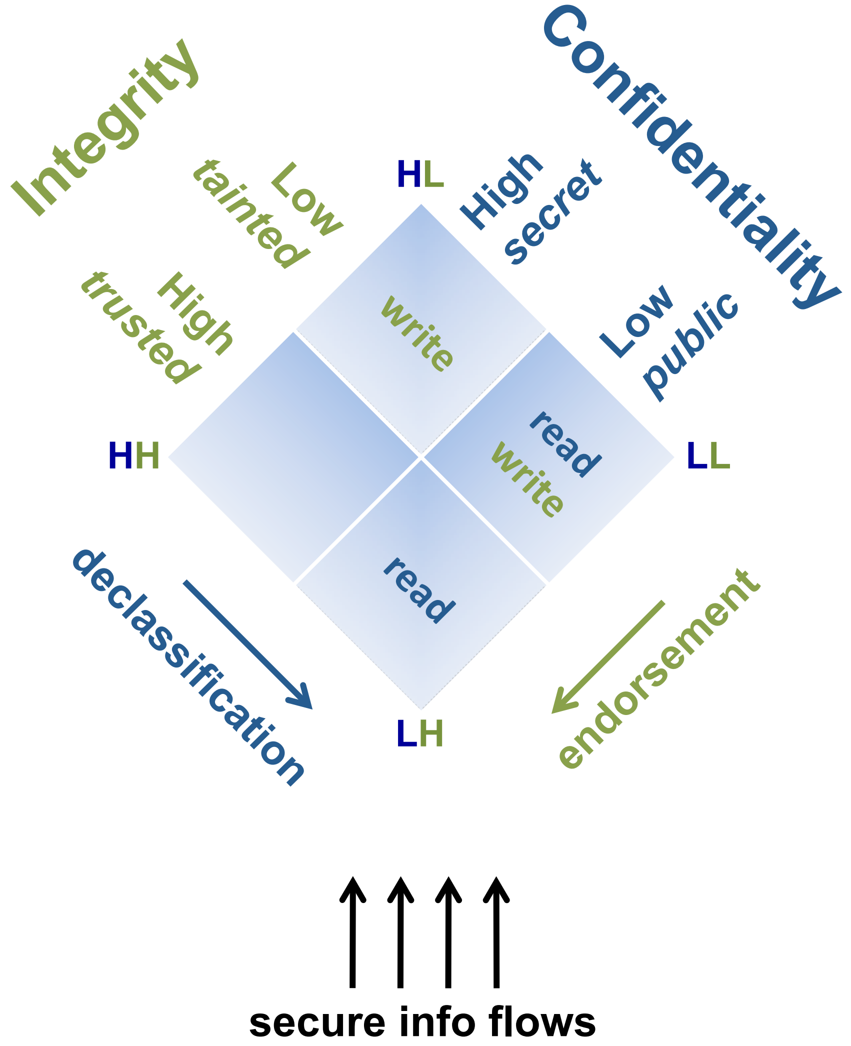
- Valid flows of information given as a security lattice of confidentiality/integrity labels (·) [Denning'76]
- Lattices enable flexible (multiparty) specification of information flow policies

  E.g. DLM [Myers Liskov'98]

- Security policies

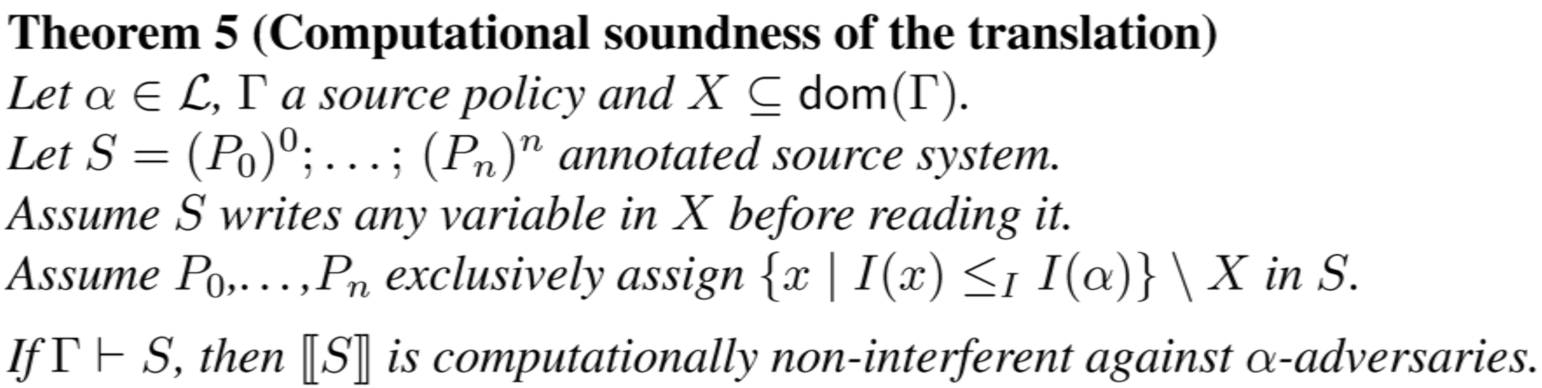
  ¡(x) level of variable *x*

  ® level of the adversary



A Typability-Preserving Compiler

- We systematically translate all accesses to selected variables *X*
- We have both functional correctness and security guarantees



Theorem 5 (Computational soundness of the translation)
Let $\alpha \in \mathcal{L}$, Γ *a source policy and* $X \subseteq \mathsf{dom}(\Gamma)$.
Let $S = (P_0)^0; \ldots; (P_n)^n$ *annotated source system.*
Assume S *writes any variable in* X *before reading it.*
Assume $P_0, \ldots, P_n$ *exclusively assign* $\{x \mid I(x) \leq_I I(\alpha)\} \setminus X$ *in* S.

If $\Gamma \vdash S$, *then* $[\![S]\!]$ *is computationally non-interferent against* α*-adversaries.*

A Typability-Preserving Compiler

- We systematically translate all accesses to selected variables *X*

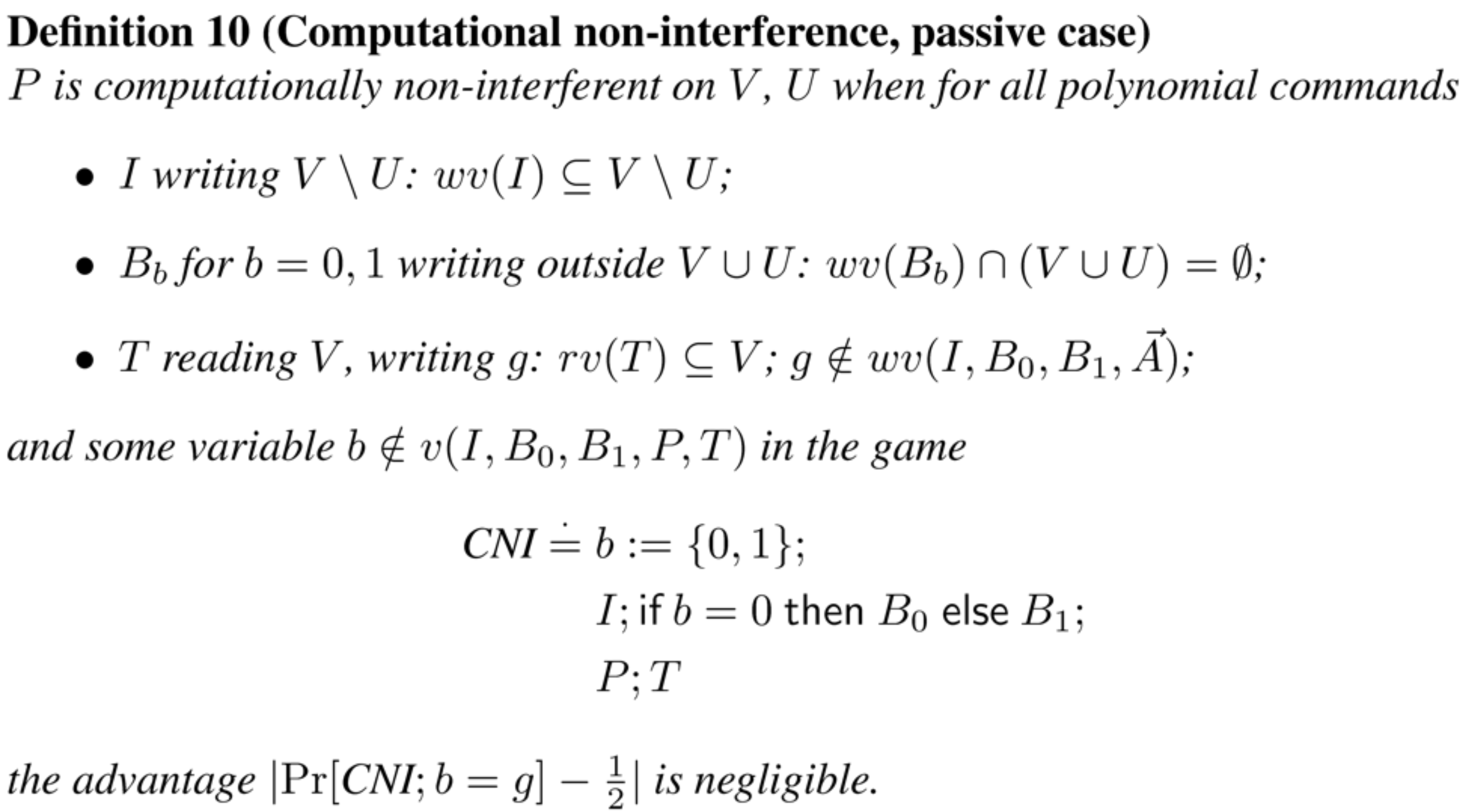
$$\begin{aligned}
\mathit{Init}_s(k_s, k_v) &= k_s, k_v := \mathcal{G}_s();\\
\mathit{Init}_e(k_e, k_d) &= k_e, k_d := \mathcal{G}_e();\\
\mathit{Read}(x \leftarrow x_e, x_s, x'_e, k_d, k_v, \mathsf{t})[P] &= \text{if } \mathcal{V}(\mathsf{t} + x_e, x_s, k_v) \text{ then}\\
&\quad\ (x'_e := x_e; x := \mathcal{D}(x'_e, k_d); P)\\
\mathit{Write}(x_s, x_e \leftarrow x, x'_e, k_e, k_s, \mathsf{t}) &= x'_e := \mathcal{E}(x, k_e);\\
&\quad\ x_s := \mathcal{S}(\mathsf{t} + x'_e, k_s); x_e := x'_e;
\end{aligned}$$

- We have both functional correctness and security guarantees

Theorem 5 (Computational soundness of the translation)
Let $\alpha \in \mathcal{L}$, Γ *a source policy and* $X \subseteq \mathsf{dom}(\Gamma)$.
Let $S = (P_0)^0; \ldots; (P_n)^n$ *annotated source system.*
Assume S *writes any variable in* X *before reading it.*
Assume $P_0, \ldots, P_n$ *exclusively assign* $\{x \mid I(x) \leq_I I(\alpha)\} \setminus X$ *in* S.

If $\Gamma \vdash S$, *then* $[\![S]\!]$ *is computationally non-interferent against* α*-adversaries.*

Computational Non-Interference



Definition 10 (Computational non-interference, passive case)
P is computationally non-interferent on V, U when for all polynomial commands

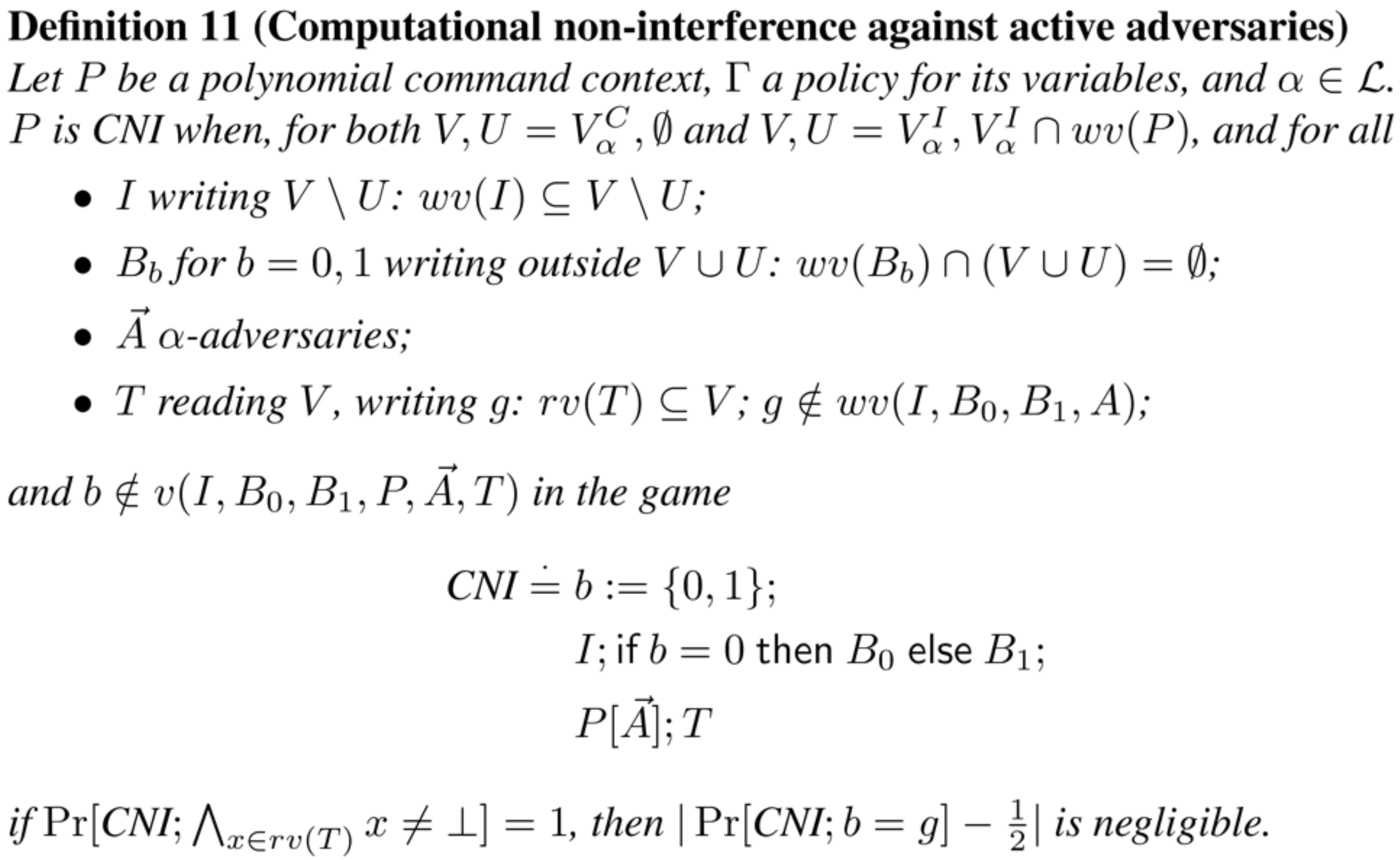
- *I writing $V \setminus U$: $wv(I) \subseteq V \setminus U$;*
- *B_b for $b = 0, 1$ writing outside $V \cup U$: $wv(B_b) \cap (V \cup U) = \emptyset$;*
- *T reading V, writing g: $rv(T) \subseteq V$; $g \notin wv(I, B_0, B_1, \vec{A})$;*

and some variable $b \notin v(I, B_0, B_1, P, T)$ in the game

$$\begin{aligned} \mathit{CNI} \doteq b &:= \{0,1\}; \\ & I; \text{if } b = 0 \text{ then } B_0 \text{ else } B_1; \\ & P; T \end{aligned}$$

the advantage $|\Pr[\mathit{CNI}; b = g] - \frac{1}{2}|$ is negligible.

Computational Non-Interference



Definition 11 (Computational non-interference against active adversaries) *Let* P *be a polynomial command context,* Γ *a policy for its variables, and* $\alpha \in \mathcal{L}$*.* P *is CNI when, for both* $V, U = V^C_\alpha, \emptyset$ *and* $V, U = V^I_\alpha, V^I_\alpha \cap wv(P)$*, and for all*

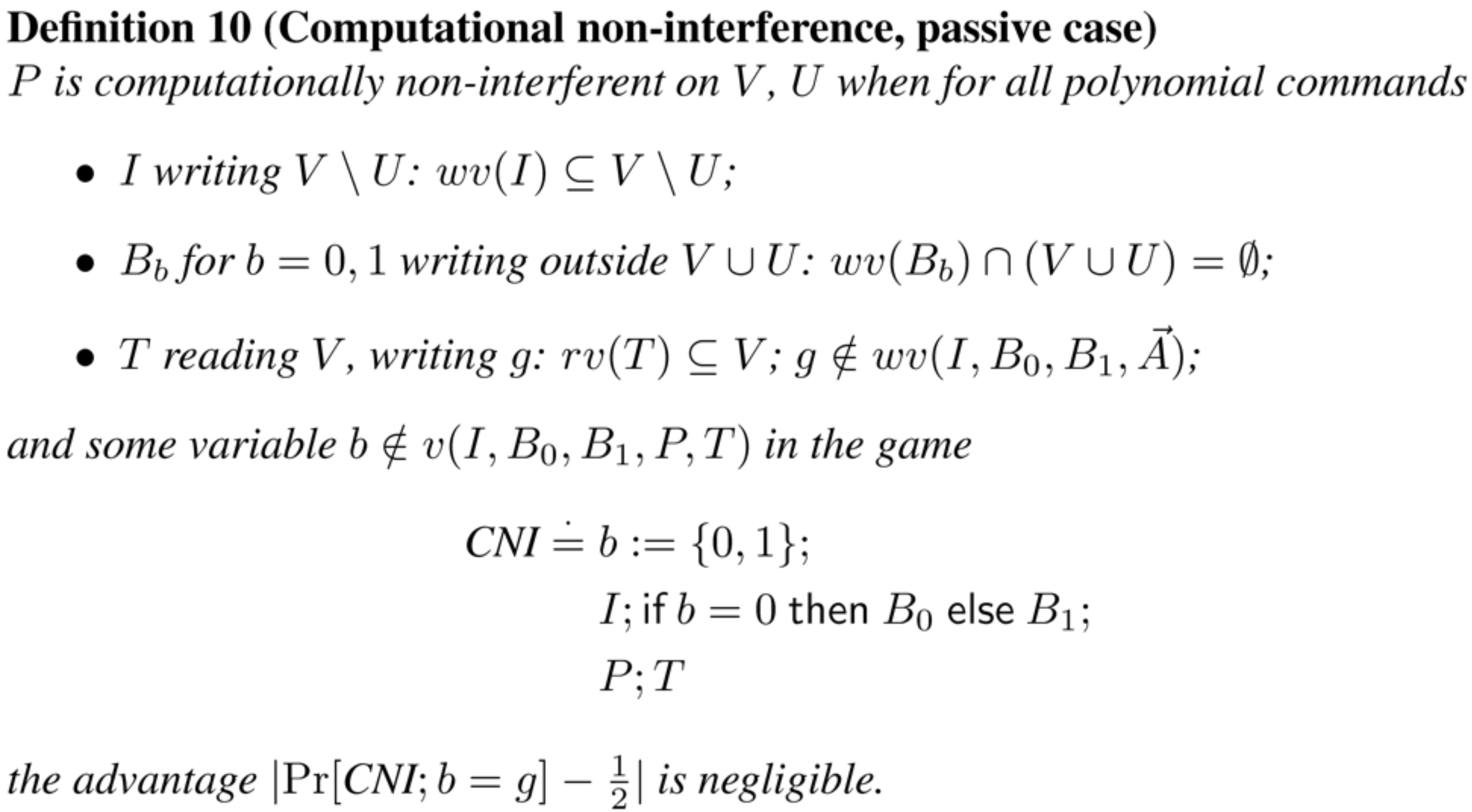
- I *writing* $V \setminus U$*:* $wv(I) \subseteq V \setminus U$*;*
- B_b *for* $b = 0, 1$ *writing outside* $V \cup U$*:* $wv(B_b) \cap (V \cup U) = \emptyset$*;*
- $\vec{A}$ α*-adversaries;*
- T *reading* V*, writing* g*:* $rv(T) \subseteq V$*;* $g \notin wv(I, B_0, B_1, A)$*;*

and $b \notin v(I, B_0, B_1, P, \vec{A}, T)$ *in the game*

$$\begin{aligned} \mathit{CNI} \doteq\ & b := \{0, 1\}; \\ & I; \mathsf{if}\ b = 0\ \mathsf{then}\ B_0\ \mathsf{else}\ B_1; \\ & P[\vec{A}]; T \end{aligned}$$

if $\Pr[\mathit{CNI}; \bigwedge_{x \in rv(T)} x \neq \bot] = 1$*, then* $|\Pr[\mathit{CNI}; b = g] - \frac{1}{2}|$ *is negligible.*

Computational Non-Interference



Definition 10 (Computational non-interference, passive case)
P is computationally non-interferent on V, U when for all polynomial commands

- *I writing $V \setminus U$: $wv(I) \subseteq V \setminus U$;*
- *B_b for $b = 0, 1$ writing outside $V \cup U$: $wv(B_b) \cap (V \cup U) = \emptyset$;*
- *T reading V, writing g: $rv(T) \subseteq V$; $g \notin wv(I, B_0, B_1, \vec{A})$;*

and some variable $b \notin v(I, B_0, B_1, P, T)$ in the game

$$\begin{aligned} \mathit{CNI} \doteq b &:= \{0,1\}; \\ &I; \mathsf{if}\ b = 0\ \mathsf{then}\ B_0\ \mathsf{else}\ B_1; \\ &P; T \end{aligned}$$

the advantage $|\Pr[\mathit{CNI}; b = g] - \frac{1}{2}|$ is negligible.

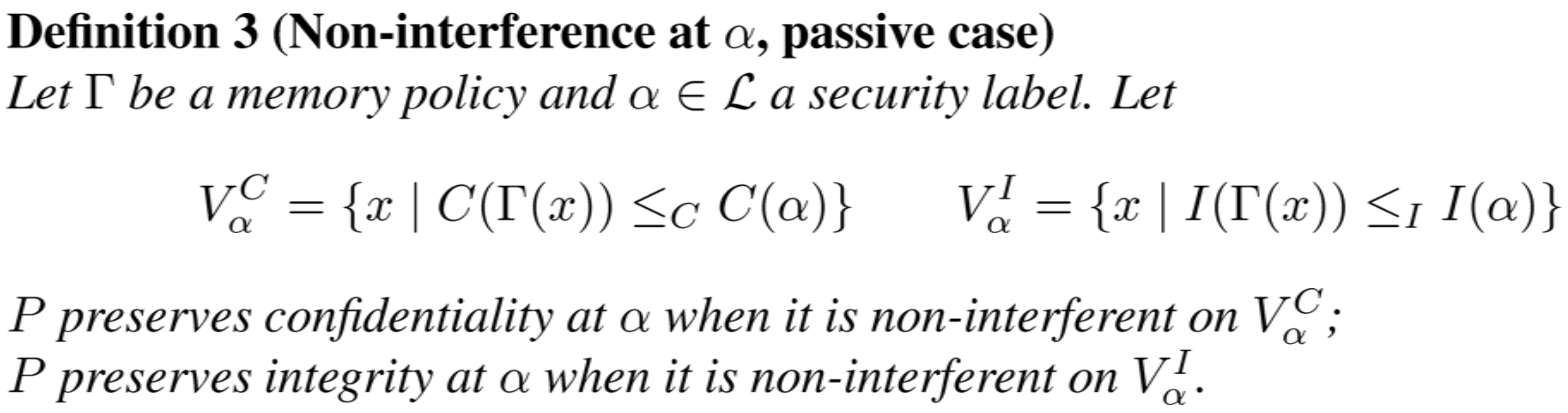
Non-Interference

Definition 1 (Memory indistinguishability)
Let V *be a set of variables.*
$\mu_0 \sim_V \mu_1$*, when* $x \in V$ *implies* $\mu_0(x) = \mu_1(x)$*.*

Definition 2 (Non-interference on V**)**
P *is non-interferent on* V *when, for all memories* μ_0 *and* μ_1*,*
if $\mu_0 \sim_V \mu_1$ *and* $\langle P, \mu_b \rangle \Downarrow \mu'_b$ *for* $b = 0, 1$*, then* $\mu'_0 \sim_V \mu'_1$*.*

Non-Interference

- We set a policy ¡ for all variables in the program
  We set a level ® for the adversary



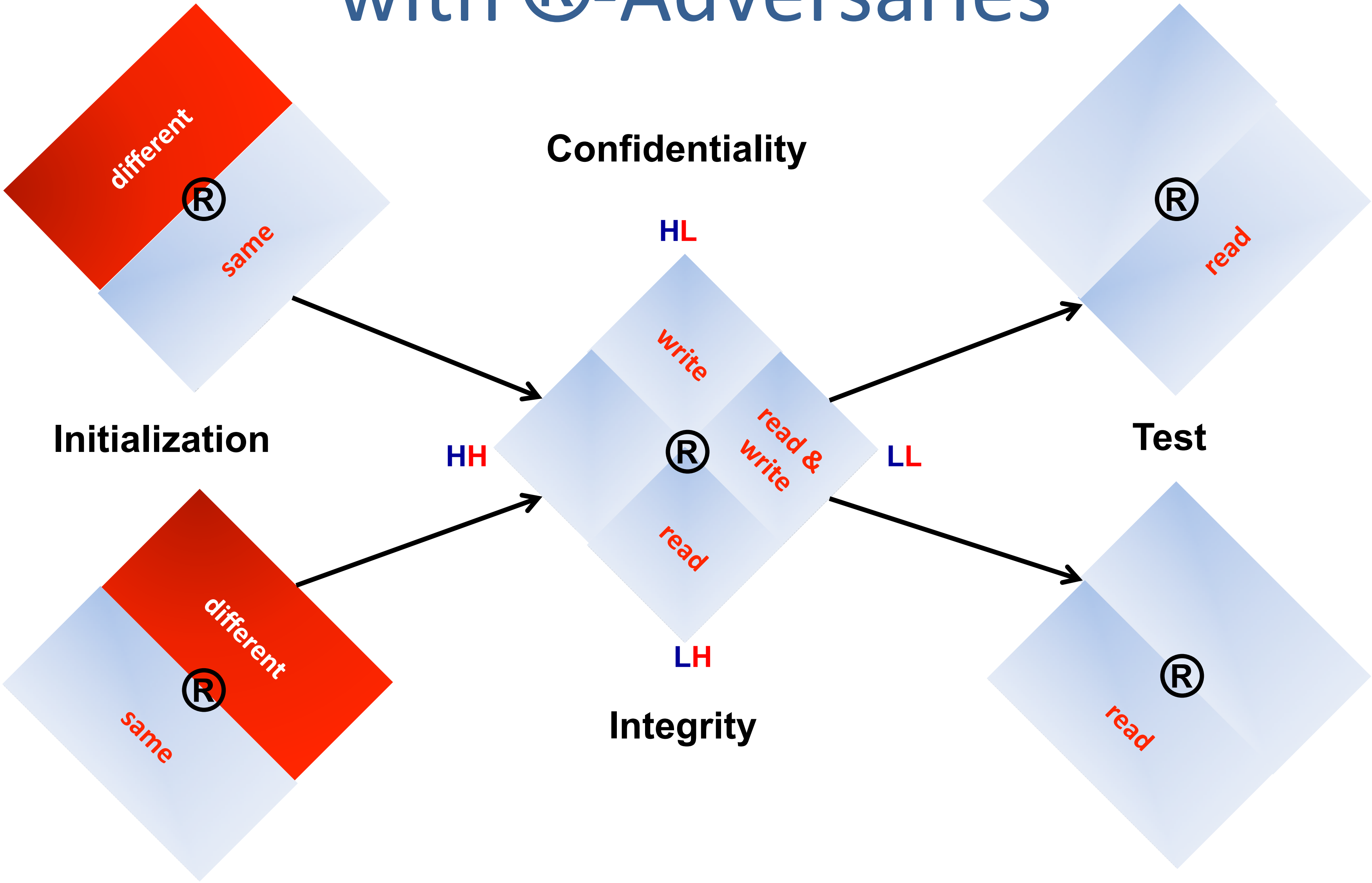
Definition 3 (Non-interference at α, passive case)
Let Γ be a memory policy and $\alpha \in \mathcal{L}$ a security label. Let

$$V_\alpha^C = \{x \mid C(\Gamma(x)) \leq_C C(\alpha)\} \qquad V_\alpha^I = \{x \mid I(\Gamma(x)) \leq_I I(\alpha)\}$$

P preserves confidentiality at α when it is non-interferent on V_α^C;
P preserves integrity at α when it is non-interferent on V_α^I.

- In the active case,
  ®-adversaries can read inside V_α^C and write outside V_α^I

Non-Interference with ®-Adversaries



Initialization

different

®

same

different

®

same

Confidentiality

HL

write

HH

®

read & write

LL

read

LH

Integrity

Test

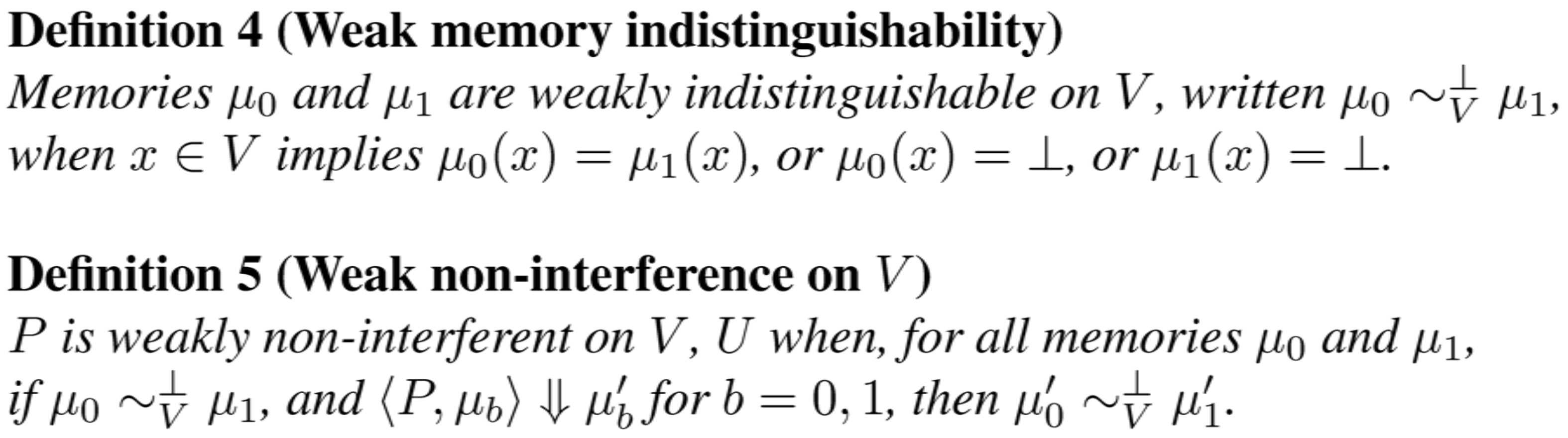
®

read

®

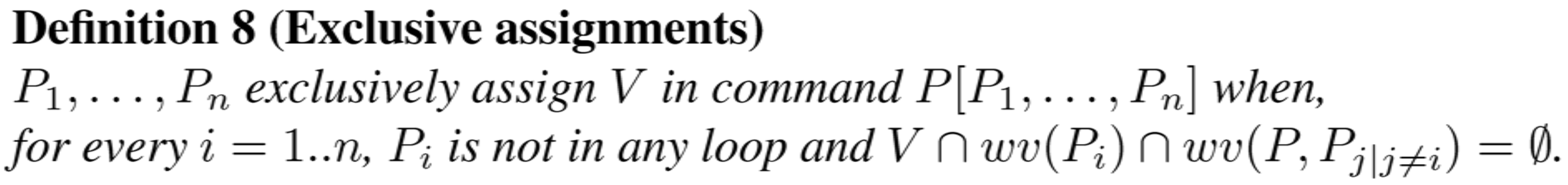
read

Non-Interference (take 2)



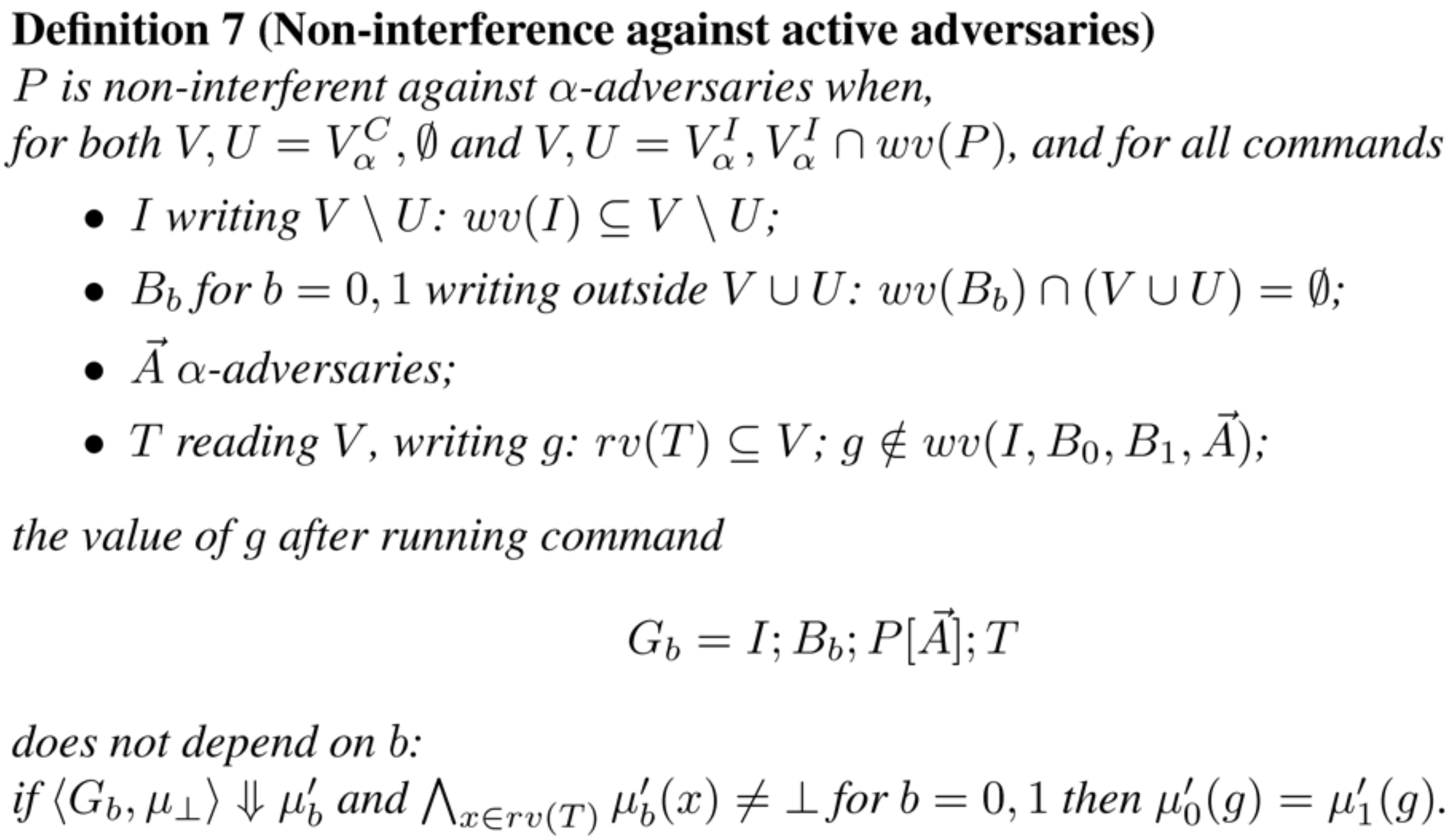
Definition 4 (Weak memory indistinguishability)
Memories μ_0 *and* μ_1 *are weakly indistinguishable on* V*, written* $\mu_0 \sim_V^\perp \mu_1$*, when* $x \in V$ *implies* $\mu_0(x) = \mu_1(x)$*, or* $\mu_0(x) = \perp$*, or* $\mu_1(x) = \perp$*.*

Definition 5 (Weak non-interference on V)
P *is weakly non-interferent on* V*,* U *when, for all memories* μ_0 *and* μ_1*, if* $\mu_0 \sim_V^\perp \mu_1$*, and* $\langle P, \mu_b \rangle \Downarrow \mu'_b$ *for* $b = 0, 1$*, then* $\mu'_0 \sim_V^\perp \mu'_1$*.*



Definition 8 (Exclusive assignments)
$P_1, \ldots, P_n$ *exclusively assign* V *in command* $P[P_1, \ldots, P_n]$ *when, for every* $i = 1..n$*,* P_i *is not in any loop and* $V \cap wv(P_i) \cap wv(P, P_{j|j \neq i}) = \emptyset$*.*

Non-Interference as a Game



Definition 7 (Non-interference against active adversaries)
P is non-interferent against α-adversaries when,
for both $V, U = V_\alpha^C, \emptyset$ and $V, U = V_\alpha^I, V_\alpha^I \cap wv(P)$, and for all commands

- *I writing $V \setminus U$: $wv(I) \subseteq V \setminus U$;*
- *B_b for $b = 0, 1$ writing outside $V \cup U$: $wv(B_b) \cap (V \cup U) = \emptyset$;*
- *$\vec{A}$ α-adversaries;*
- *T reading V, writing g: $rv(T) \subseteq V$; $g \notin wv(I, B_0, B_1, \vec{A})$;*

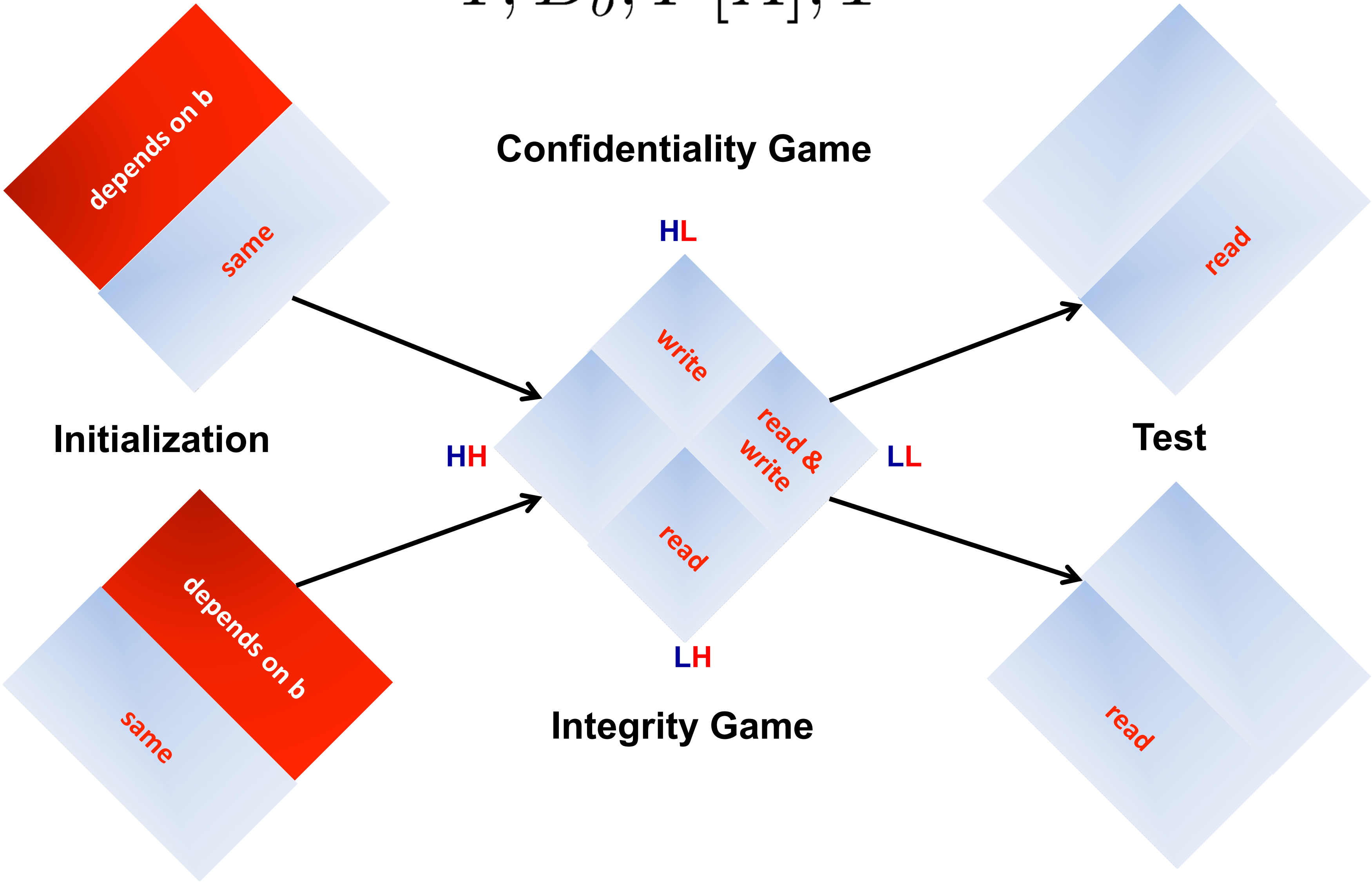
the value of g after running command

$$G_b = I; B_b; P[\vec{A}]; T$$

does not depend on b:
if $\langle G_b, \mu_\perp \rangle \Downarrow \mu'_b$ and $\bigwedge_{x \in rv(T)} \mu'_b(x) \neq \perp$ for $b = 0, 1$ then $\mu'_0(g) = \mu'_1(g)$.

Non-Interference as a Game

$$I; B_b; P[\vec{A}]; T$$



Confidentiality Game

Initialization

Test

Integrity Game

Old Intro/General Slides

Crypto Colloquium, October 2008

A Cryptographic Compiler for Information-Flow Security

Cédric Fournet

Programming principles and tools
Microsoft Research, Cambridge

Tamara Rezk Gurvan le Guernic

Secure distributed computations
MSR—INRIA joint centre, Orsay

http://www.msr-inria.inria.fr/projects/sec

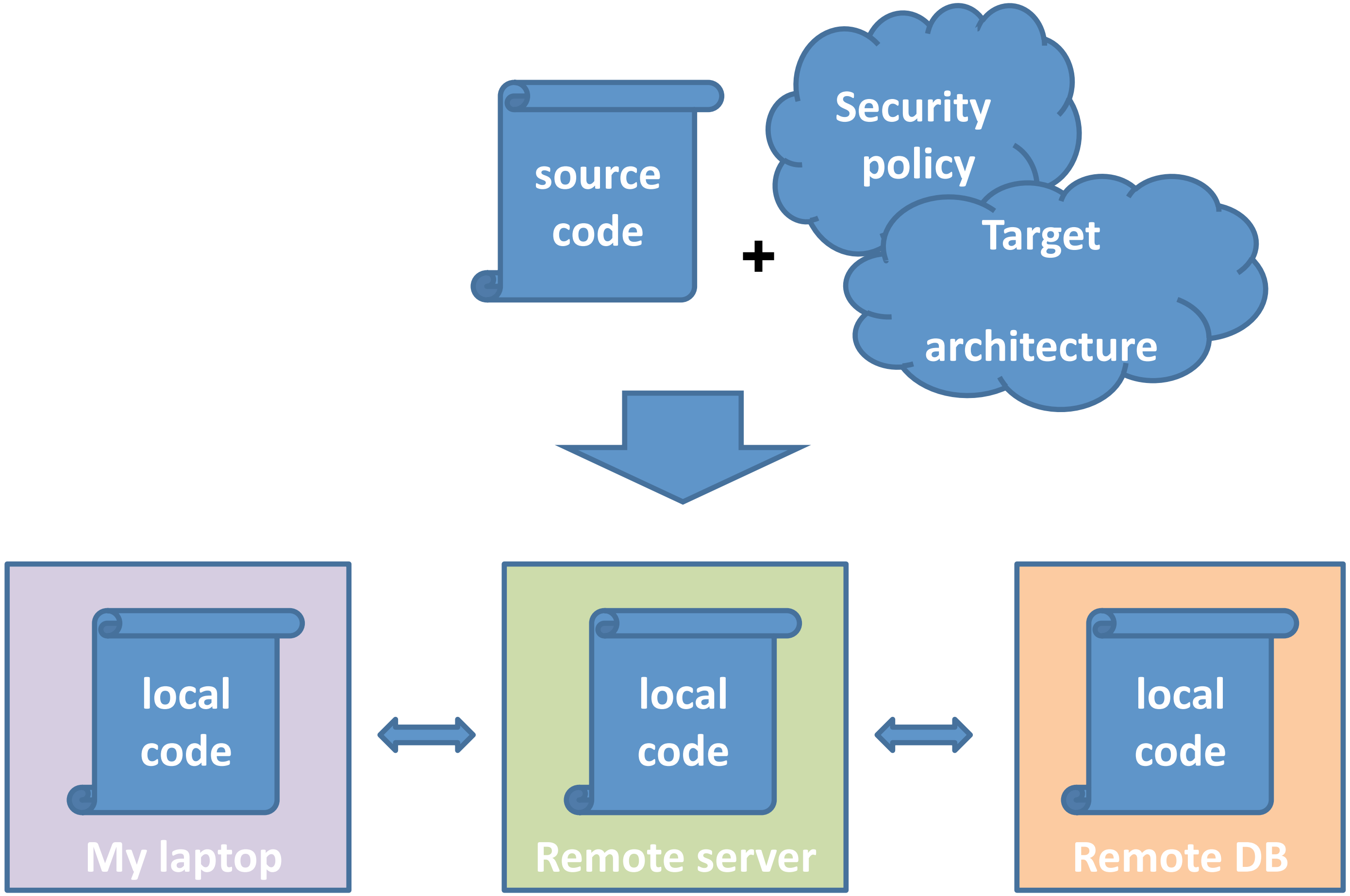
Writing secure code?

- Security relies on a precise mapping from goals to mechanisms, but
  - high-level security goals are often left informal
  - low-level enforcement mechanisms are complex and hidden in system implementations (cryptography, network stack)
- Programming frameworks don't help much
  - Language designs and implementations predate security concerns
  - Implicit trust in the execution environment (TCB)
    - What if remote hosts are corrupted?

Provable Cryptography

- Cryptography may help, but...
  - Interesting properties (privacy, integrity) depend on the application
    - These properties must be clear to the programmer
  - Modern applications dynamically select and reconfigure their protocols
    - e.g. Web services, grid computing; flexibility is great but not for security
  - Hence, experts can’t verify security without knowing the application
    - Transparent security is a myth

Information-Flow between Partially-Trusted Hosts


source code

+

Security policy

Target architecture

local code

My laptop

local code

Remote server

local code

Remote DB

Motivation and Goals

- Need for simple programming language abstractions for security and their robust crypto implementation
- Need for stronger connections between high-level security goals and the usage of crypto protocols

- A compiler that implements cryptographic and distribution issues (transparent to the programmer)
- The programmer specifies a high-level security policy (confidentiality and integrity of data)
- If the source program is typable for one policy, our compiler generates low-level, well-typed cryptographic code

Related Work

- Non-interference
  Goguen & Meseguer 82, Bell & LaPadula 76, Denning 76
- Declassification
  Principles and dimensions by Sabelfeld & Sands 05
  Robust Declassification Zdancewic & Myers 01
  Enforcing Robust Declassification Myers, Sabelfeld, Zdancewic 04
- Secure information flow and Cryptography
  Laud 01, Backes & Pfitzmann 02 03
- Secure implementations
  Jif Split Myers & Zheng et al 01
- Cryptographic DLM
  Vaughan & Zdancewic 07

Two Models for Cryptography

- Models are needed to design and analyze protocols; models may hide important flaws of real systems.
- Two approaches have been successfully applied to protocols and programs that use cryptography

  Formal, or symbolic approach (Needham-Schroeder, Dolev-Yao, ... late 70's)
  - Structural view of protocols, using simple formal languages, and methods from logic, programming languages, concurrency
  - Compositional, good tools, scales to large systems (IPSEC, Web Services)
  - Too abstract?

  Computational approach (Yao, Goldwasser, Micali, Rivest, ... early 80's)
  - Concrete view: messages are ensembles of bitstrings
  - Adversaries range over probabilistic Turing Machines
  - More accurate, more widely accepted
  - Delicate (informal) reduction proofs; scalability issues

Formal Computational Cryptography

- How to get the best of both worlds?
  - We need some combination to verify large crypto-based systems
  - Can we carry over results and tools from one model to the other?

  Soundness property (desired)

  "If a security property can be proved in a formal model,
  then it holds in a computational model"

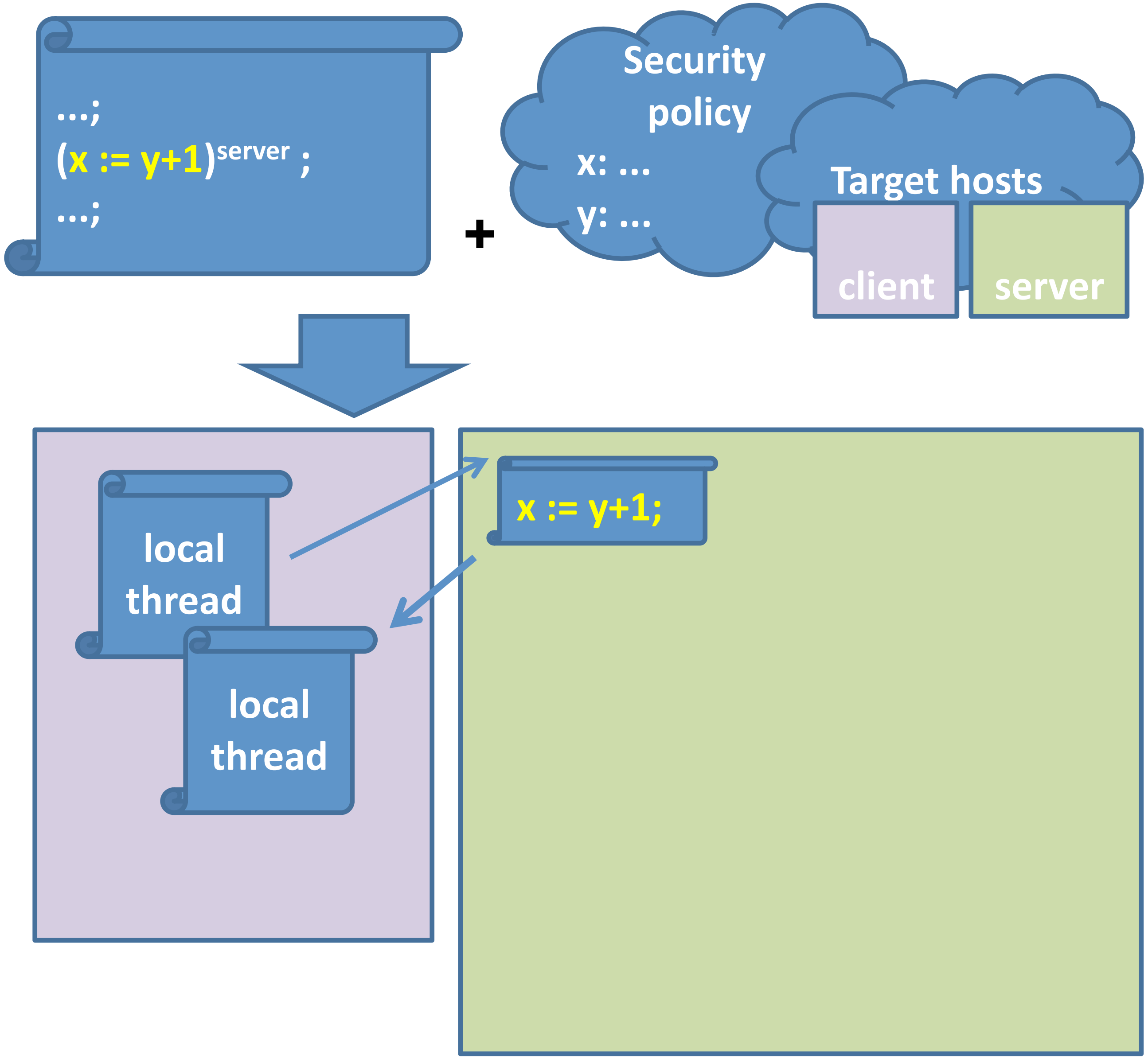
- Computational soundness of formal cryptography
  - Active research, both positive and negative results
- Instead, we design **cryptographically sound abstractions** for **high-level security programming**
  - For languages, type systems, communications, sessions...
  - Not directly for formal cryptography:
    We care about security properties (not implementation details)

Information-Flow Security

- Information flow provides a clean specification of security
  - Secret inputs do not leak to public outputs
  - Tainted inputs do not influence trusted outputs

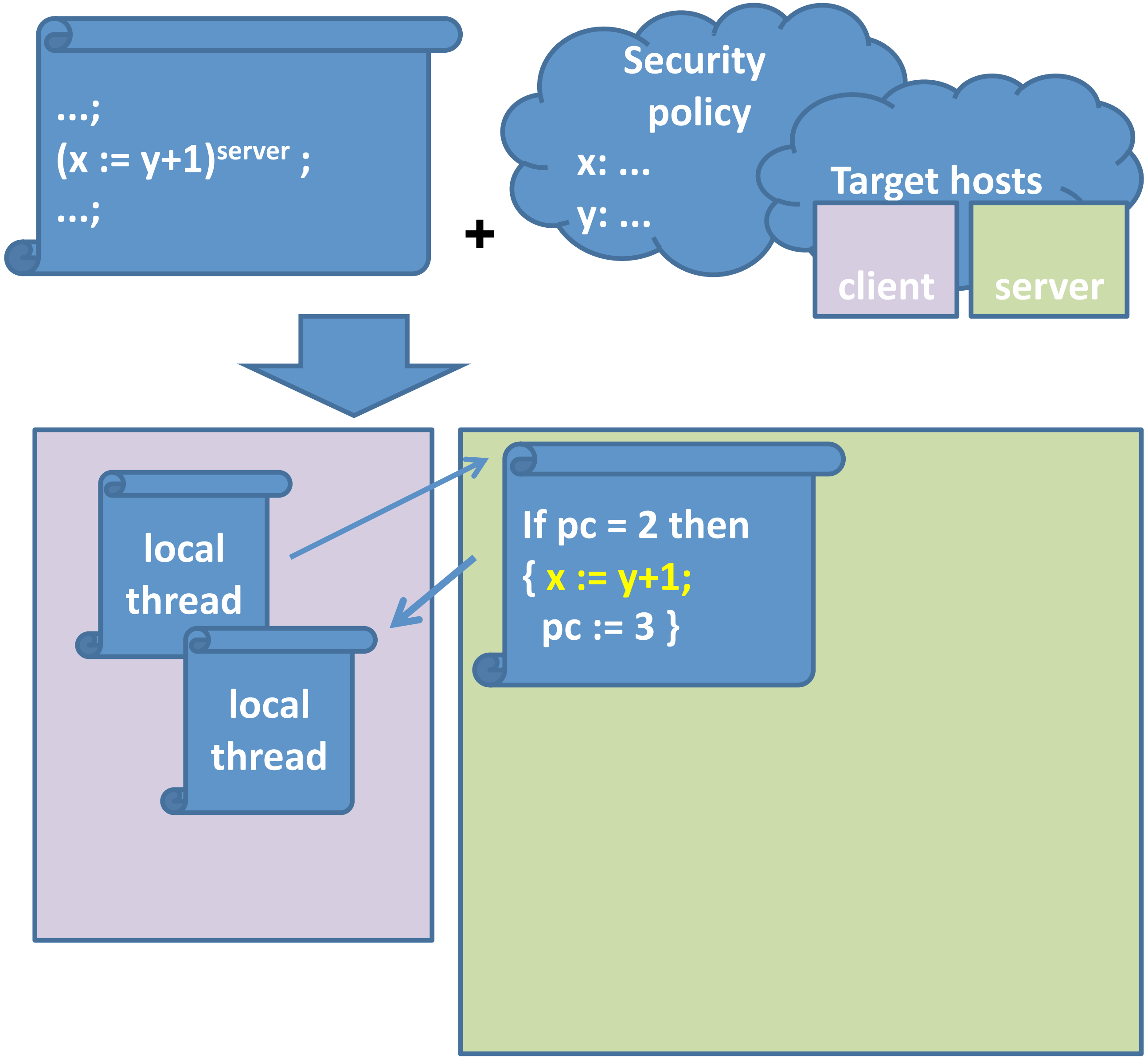
  ... but its enforcement in concrete systems is delicate
- We compile **imperative programs with information-flow policies** down to **cryptographic (probabilistic) distributed programs**
  - Secrecy by encryptions, integrity by digital signatures
- Soundness relies on a new type system
  - Types capture mutual dependencies between secrecy and integrity levels for all keys and payloads used in our code
  - Well-typed programs are **computationally non-interferent**:
    probabilistic polynomial-time adversaries
    gain illegal information only with negligible probability

Cryptographic Compilation



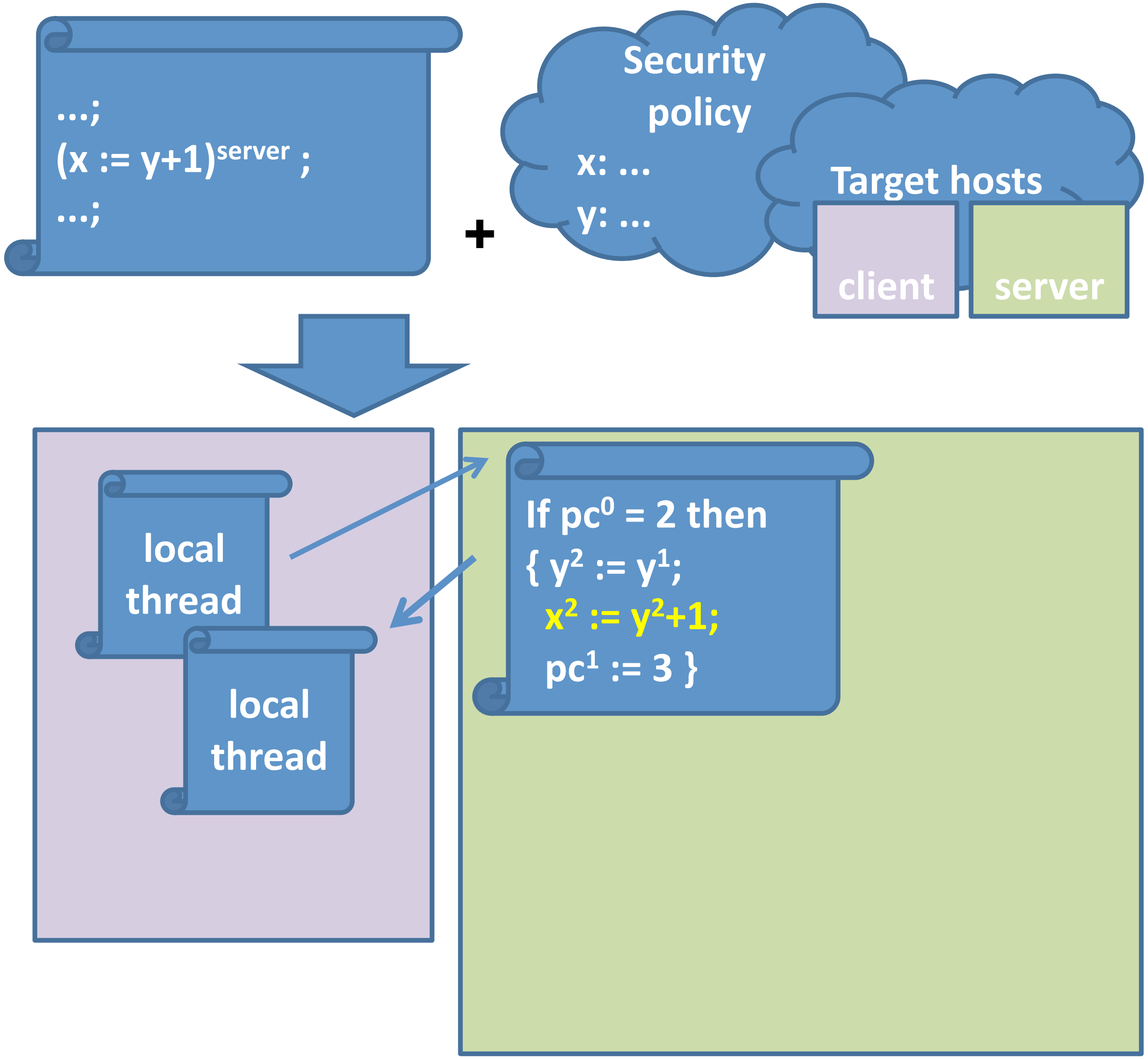
1. Split the program into local threads
    - explicit control flow between machines
2. Secure control flow, using program counters (shared, high integrity)
3. Split shared variables into local replicas (single, static assign)
4. Cryptographically protect reads and writes
    - encrypt and/or sign
    - use auxiliary keys
    - use long-lived PKI for bootstrapping
5. Verify our code against extended policy
    - all shared variables are public & tainted
    - except for long-term verification keys

Cryptographic Compilation



1. Split the program into local threads
   - explicit control flow between machines
2. Secure control flow, using program counters (shared, high integrity)
3. Split shared variables into local replicas (single, static assign)
4. Cryptographically protect reads and writes
   - encrypt and/or sign
   - use auxiliary keys
   - use long-lived PKI for bootstrapping
5. Verify our code against extended policy
   - all shared variables are public & tainted
   - except for long-term verification keys

Cryptographic Compilation



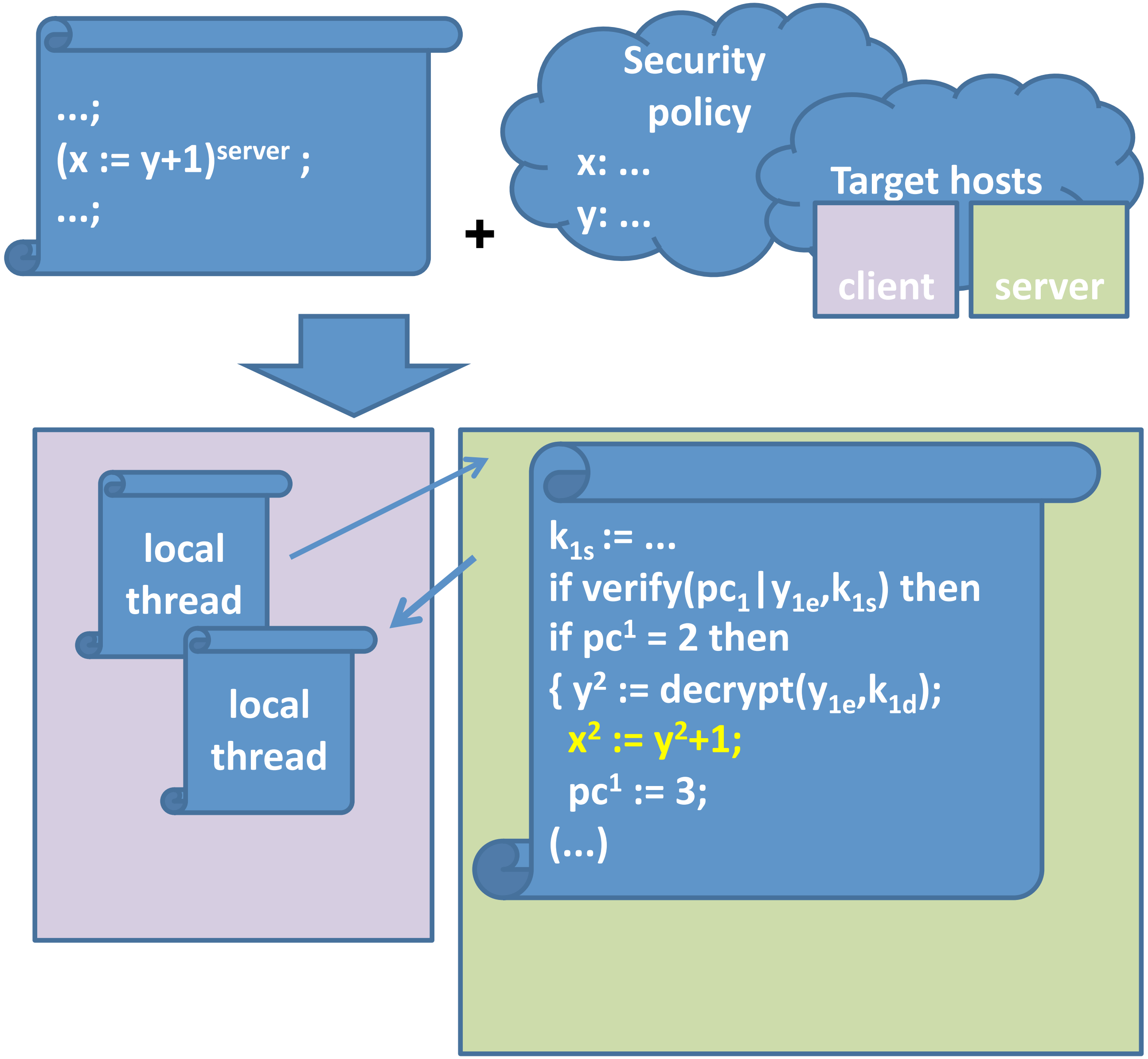
…;
$(x := y+1)^{server}$;
…;

\+

If $pc^0 = 2$ then
$\{\ y^2 := y^1;$
$x^2 := y^2+1;$
$pc^1 := 3\ \}$

1. Split the program into local threads
   - explicit control flow between machines
2. Secure control flow, using program counters (shared, high integrity)
3. Split shared variables into local replicas (single, static assign)
4. Cryptographically protect reads and writes
   - encrypt and/or sign
   - use auxiliary keys
   - use long-lived PKI for bootstrapping
5. Verify our code against extended policy
   - all shared variables are public & tainted
   - except for long-term verification keys

Cryptographic Compilation



```
…;
$(x := y+1)^{server}$ ;
…;
```

+

Security policy

x: …

y: …

Target hosts

client

server

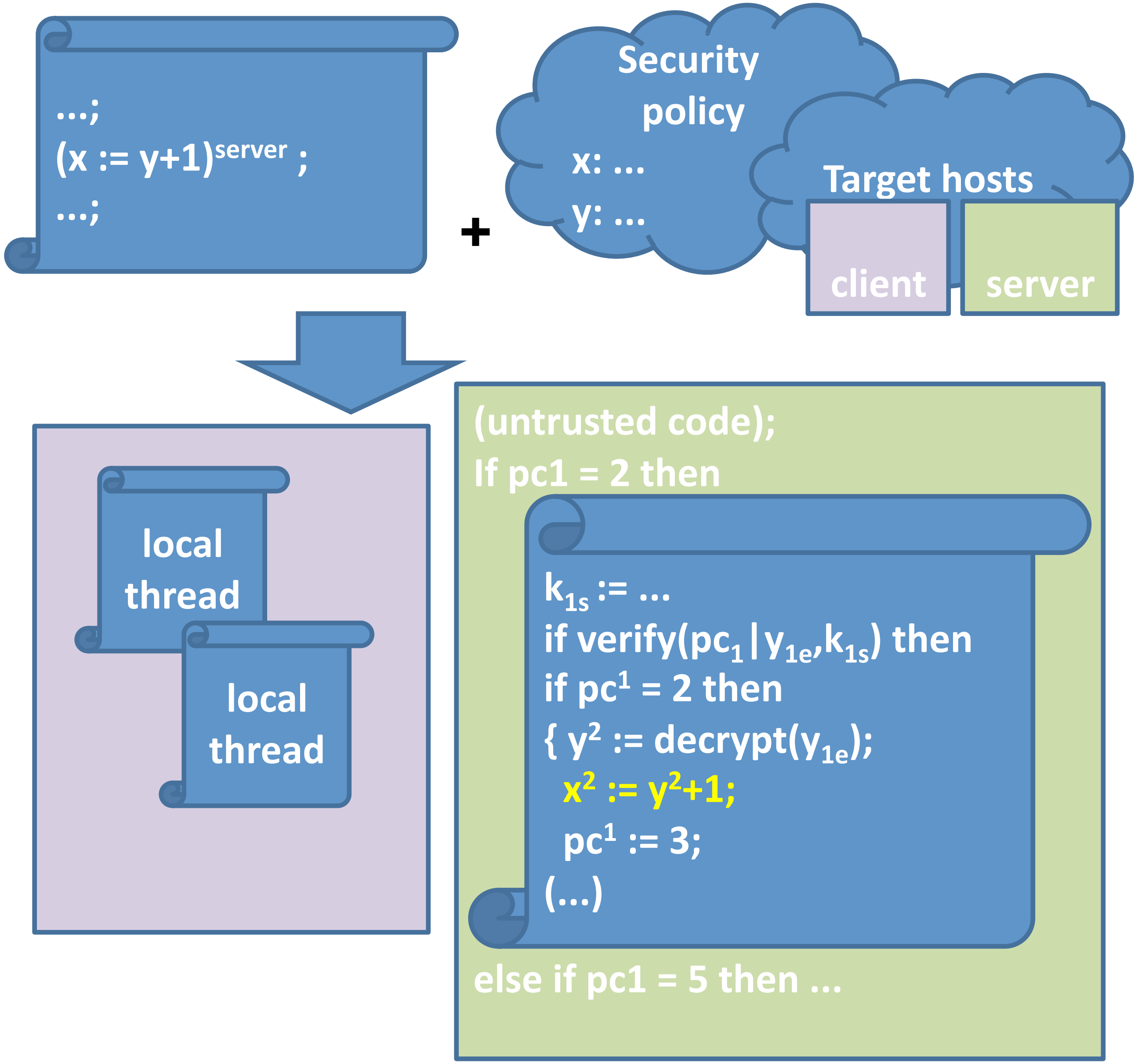
local thread

local thread

```
$k_{1s}$ := …
if verify($pc_1 | y_{1e}, k_{1s}$) then
if $pc^1$ = 2 then
{ $y^2$ := decrypt($y_{1e}, k_{1d}$);
  $x^2 := y^2+1$;
  $pc^1$ := 3;
(…)
```

1. Split the program into local threads
   - explicit control flow between machines
2. Secure control flow, using program counters (shared, high integrity)
3. Split shared variables into local replicas (single, static assign)
4. Cryptographically protect reads and writes
   - encrypt and/or sign
   - use auxiliary keys
   - use long-lived PKI for bootstrapping
5. Verify our code against extended policy
   - all shared variables are public & tainted
   - except for long-term verification keys

Cryptographic Compilation



…;

(x := y+1)^{server} ;

…;

\+

Security policy

x: …

y: …

Target hosts

client

server

local thread

local thread

(untrusted code);

If pc1 = 2 then

k_{1s} := …

if verify(pc_1|y_{1e},k_{1s}) then

if pc^1 = 2 then

{ y^2 := decrypt(y_{1e});

x^2 := y^2+1;

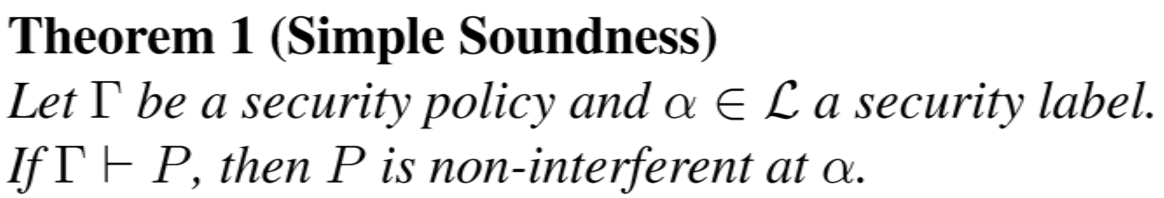
pc^1 := 3;

(…)

else if pc1 = 5 then …

1. Split the program into local threads
   - explicit control flow between machines
2. Secure control flow, using program counters (shared, high integrity)
3. Split shared variables into local replicas (single, static assign)
4. Cryptographically protect reads and writes
   - encrypt and/or sign
   - use auxiliary keys
   - use long-lived PKI for bootstrapping
5. Verify our code against extended policy
   - all shared variables are public & tainted
   - except for long-term verification keys

Typing Expressions and Commands



Theorem 1 (Simple Soundness)
Let Γ *be a security policy and* $\alpha \in \mathcal{L}$ *a security label.*
If $\Gamma \vdash P$*, then* P *is non-interferent at* α*.*

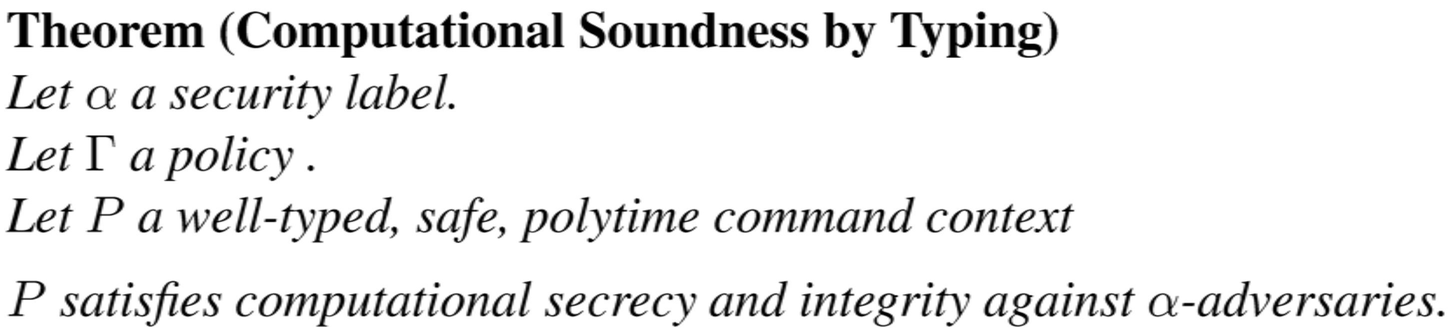
Typing Command Contexts

Theorem 2
Let Γ *be a policy and* $\alpha \in \mathcal{L}$ *a security label.*
Assume $\Gamma \vdash P$ *and*
all P' *in* $(\mathsf{if}\ e\ \mathsf{then}\ P')$ *typed by* CHECK *exclusively assign* V_α^I.

The command context P *satisfies non-interference against* α*-adversaries.*

Typing Probabilistic Programs

- We develop a type system for command contexts
  with rules for probabilistic functions and cryptography



Theorem (Computational Soundness by Typing)
Let α a security label.
Let Γ a policy .
Let P a well-typed, safe, polytime command context

P satisfies computational secrecy and integrity against α-adversaries.

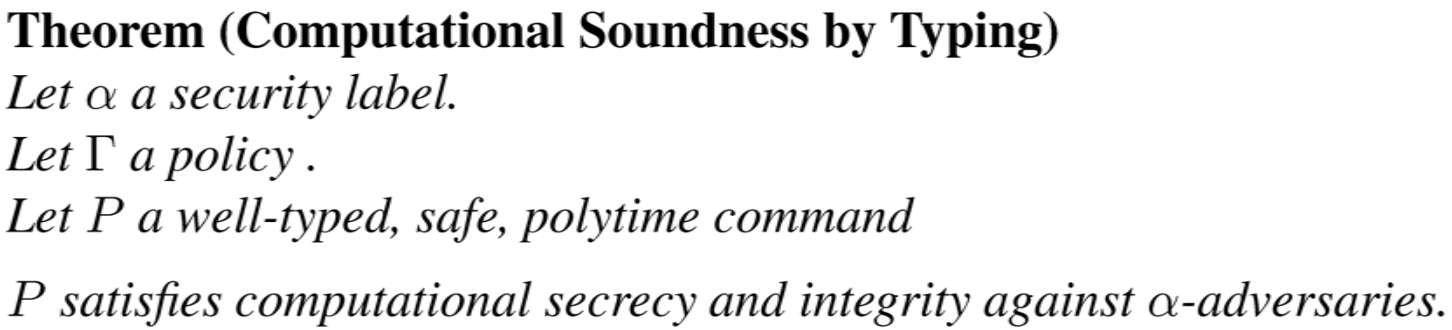
- The proof is by a series of typed program transformations (games)

Cryptographic Types

$\tau ::=$	$t(\ell)$	Security types
$t ::=$	$\mathsf{Data} \mid t * t$	Data types for payloads
$\mid$	$\mathsf{Enc}\,\tau\,K \mid \mathsf{Ke}\,\tau\,K \mid \mathsf{Kd}\,\tau\,K$	Data types for asymmetric encryption
$\mid$	$\mathsf{SEnc}\,\tau\,K \mid \mathsf{Ked}\,\tau\,K$	Data types for symmetric encryption
$\mid$	$\mathsf{Sig}\,\tau \mid \mathsf{Ks}\,\mathsf{F}\,K \mid \mathsf{Kv}\,\mathsf{F}\,K$	Data types for signing
$\mid$	$\mathsf{Mac}\,\tau \mid \mathsf{Km}\,\mathsf{F}\,K$	Data types for keyed hashes

- By design, these types suffice to build efficient protocols, including key establishment and selective key reuse.
  - Our types keep track of static names *K* for keys, of tags for signing (*F: t* ↦ *¿),* and of maximal message lengths.
  - Our typing rules capture computationally sound patterns of declassifications and endorsement

Security by Typing [POPL'08]

- 

  Theorem (Computational Soundness by Typing)
  Let α *a security label.*
  Let Γ *a policy .*
  Let P *a well-typed, safe, polytime command*

  P *satisfies computational secrecy and integrity against* α*-adversaries.*

  - the proof is by a series of typed program transformations (games)

- Starting from well-typed source programs,
  the compiler yields well-typed cryptographic code
  for an extension of the source program policy, hence
- Compilation preserves all information-flow properties:
  - an adversary that interacts with high-level code and entirely controls low-level code gains illegal information only with negligible probability.

Typing rules for Signatures

$$\textsc{GenS}\quad \frac{\Gamma(k_s) = \mathsf{Ks\,F}\,K(\ell_s) \qquad \Gamma(k_v) = \mathsf{Kv\,F}(\ell_k)}{\vdash k_s, k_v := \mathcal{G}_s() \;:\; \ell_s \sqcap \ell_k}$$

$$\textsc{Sig}\quad \frac{\begin{array}{c}\Gamma(k_s) = \mathsf{Ks\,F}\,K(\ell_s) \qquad \mathsf{F}(\mathsf{t}) = \tau \qquad \Gamma(x) = \mathsf{Sig}\,\tau(\ell_x) \\ \vdash m : \tau \qquad L(\tau) \leq L(x) \qquad I(\ell_s) \leq_I I(x)\end{array}}{\vdash x := \mathcal{S}(\mathsf{t} + m, k_s) \;:\; \ell_x}$$

$$\textsc{Ver}\quad \frac{\begin{array}{c}\Gamma(k_v) = \mathsf{Kv\,F}(\ell_k) \qquad \mathsf{F}(\mathsf{t}) = \tau \qquad \Gamma(x) = \tau \\ \vdash v : \tau' \qquad \vdash m : \mathsf{Sig}\,\tau(\ell_m) \qquad \vdash P : \ell_P \\ C(\ell_m) \sqcup C(\tau') \leq_C C(x) \qquad \ell_k \leq L(x)\end{array}}{\vdash \mathsf{if}\ \mathcal{V}(\mathsf{t} + v, m, k_v)\ \mathsf{then}\ (x := v; P) \;:\; L(x)}$$

Security by Typing

- [POPL'08] Starting from well-typed source programs,
  the compiler yields well-typed cryptographic code
  for an extension of the source program policy
- Hence, compilation preserves all information-flow properties:
  - **an adversary that interacts with our compiled code and entirely controls low-level code gains illegal information only with negligible probability.**

- We now have similar guarantees for "insecure" source programs
  - **an adversary that interacts with our compiled code and entirely controls low-level code does not gain (much) more information than an adversary that interacts with source code.**