Modular Verification of Secure and Leakage-Free Systems:

From Application Specification to Circuit-Level Implementation





Anish Athalye[1], Henry Corrigan-Gibbs[1], M. Frans Kaashoek[1], Joseph Tassarotti[2], Nickolai Zeldovich[1]

[1] MIT CSAIL [2] New York University

Problem: bugs across the software and hardware stack

Software bugs

Hardware bugs

Timing side channels

Problem: bugs across the software and hardware stack

Software bugs

Hardware bugs

Timing side channels

STM32F303xB/C — Description of device errata

2.2.2 Data Read when the CPU accesses successively SRAM address “A” and SRAM address “A + offset of 16 KBytes (0x4000)”

Description

If the CPU writes to an address A in the SRAM memory and immediately (the cycle after) reads an address B in the SRAM memory, while B = A+0x4000, the read operation will return the content at address A instead of the content of address B.

CVE-ID	
CVE-2019-18672	Learn more • CVSS Severity Mappings • CPE

Description

Insufficient checks in the finite state machine o 6.2.2 allow a partial reset of cryptographic secr breaks the security of U2F for new server regis vulnerability can be exploited by unauthenticat

TPM-FAIL: TPM meets Timing and Lattice Attacks

Daniel Moghimi and Berk Sunar, *Worcester Polytechnic Institute, Worcester, MA, USA;* Thomas Eisenbarth, *University of Lübeck, Lübeck, Germany;* Nadia Heninger, *University of California, San Diego, CA, USA*

https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Nitrokey / nitrokey-pro-firmware Public — Watch 16

Code | Issues 37 | Pull requests 6 | Actions | Projects

Fix off by one error in OTP slot range check

Merged — szszszsz merged 1 commit into Nitrokey:master from FlorianUekermann

Security advisory YSA-2018-01 – Security issue with password pro applet on YubiKey NEO

Published date: **2018-01-16**

Tracking IDs: **YSA-2018-01**

Summary

Oscar Mira and Roi Martin from the Schibsted security team informed us of a se Open Authentication) applet on the YubiKey NEO. The YubiKey OATH applet is password (TOTP) and HMAC-based one-time password (HOTP) codes that are t Authenticator app. To provide an extra layer of protection against unauthorize applet can be protected with an optional password; a feature unique to the Yu password (OTP) code generators. The issue may allow an individual in physica remove the password protection of the OATH applet and view the TOTP/HOTP companion Yubico Authenticator app, without knowing the password.

CVE-ID	
CVE-2019-18671	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

Insufficient checks in the USB packet handling of the ShapeShift KeepKey hardware wallet before firmware 6.2.2 allow out-of-bounds writes in the .bss segment via crafted messages. The vulnerability could allow code execution or other forms of impact. It can be triggered by unauthenticated attackers and the interface is reachable via WebUSB.

CVE-ID	
CVE-2021-31	

Description

Insufficient length checks in the ShapeShift KeepKey hardware wallet firm buffer overflow via crafted messages. The overflow in ethereum_extractTh can circumvent stack protections and lead to code execution. The vulnerab over WebUSB.

CVE-ID	
CVE-2018-6875	Learn more at National Vulnerability Database (NVD) • CVSS Severity Rating • Fix Information • Vulnerable Software Versions • SCAP Mappings • CPE Information

Description

Format String vulnerability in KeepKey version 4.0.0 allows attackers to trigger information display (of information that should not be accessible), related to text containing characters that the device's font lacks.

SecurityAdvisory 2015-04-14

Tracking IDs: YSA-2015-1 and CVE-2015-3298.

Summary

Minerva: The curse of ECDSA nonces

Systematic analysis of lattice attacks on noisy leakage of bit-length of ECDSA nonces

Ján Jančár[1], Vladimír Sedláček[1,2], Petr Švenda[1] and Marek Sýs[1]

[1] Masaryk University,
[2] Ca' Foscari University of Venice
{j08ny,vlada.sedlacek}@mail.muni.cz;{svenda,syso}@fi.muni.cz

Abstract. We present our discovery[1] of a group of side-channel vulnerabilities in implementations of the ECDSA signature algorithm in a widely used Atmel AT90SC FIPS 140-2 certified smartcard chip and five cryptographic libraries (libgcrypt, wolfSSL, MatrixSSL, SunEC/OpenJDK/Oracle JDK, Crypto++). Vulnerable implementations leak the bit-length of the scalar used in scalar multiplication via timing. Using leaked bit-length, we mount a lattice attack on a 256-bit curve, after observing enough signing operations. We propose two new methods to recover the full private key requiring just 500 signatures for simulated leakage data, 1200 for real cryptographic

Bugs across the stack: timing

Bugs across the stack: timing

Non-constant-time code

```
bool check(char *password, char *guess) {
  for (int i = 0; i < PW_LEN; i++) {
    if (password[i] != guess[i]) {
      return false;
    }
  }
  return true;
}
```

Bugs across the stack: timing

Non-constant-time code

Compiler-introduced timing issues

```
bool check(char *password, char *guess) {
  bool eq = true;
  for (int i = 0; i < PW_LEN; i++) {
    eq &= (password[i] == guess[i]);
  }
  return eq;
}
```

Bugs across the stack: timing

Non-constant-time code

Compiler-introduced timing issues

```
bool check(char *password, char *guess) {
  bool eq = true;
  for (int i = 0; i < PW_LEN; i++) {
    eq &= (password[i] == guess[i]);
  }
  return eq;
}
```

Breaking Bad: How
Compilers Break Constant-Time Implementations

Moritz Schneider
ETH Zurich

Daniele Lain
ETH Zurich

Ivan Puddu
ETH Zurich

Nicolas Dutly
ETH Zurich

Srdjan Čapkun
ETH Zurich

***Abstract*—The implementations of most hardened cryptographic libraries use defensive programming techniques for side-channel resistance. These techniques are usually specified as guidelines to developers on specific code patterns to use or avoid. Examples include performing arithmetic operations to choose between two variables instead of executing a secret-dependent branch. However, such techniques are only meaningful if they persist across compilation. In this paper, we investigate how optimizations used by modern compilers break the protections in-**

be deployed everywhere, leaving less vetted architectures as second-class citizens in terms of security and hence potentially more susceptible to attacks. A solution to this problem is to compile the portable source code of security-critical libraries with special compilers that automatically remove side channels [12], [38]. However, these compilers suffer from a set of shortcomings: support for processor architectures is

Bugs across the stack: timing

Non-constant-time code

Compiler-introduced timing issues

Microarchitectural side channels

2019 IEEE Symposium on Security and Privacy

Spectre Attacks: Exploiting Speculative Execution

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4],
Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Moritz Lipp[5],
Stefan Mangard[5], Thomas Prescher[6], Michael Schwarz[5], Yuval Yarom[8]
[1] Independent (www.paulkocher.com), [2] Google Project Zero,
[3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Mar
[5] Graz University of Technology, [6] Cyberus Technology,
[7] Rambus, Cryptography Research Division, [8] University of Adelaide and Data6

***Abstract*—Modern processors use branch prediction and speculative execution to maximize performance. For example, if the destination of a branch depends on a memory value that is in the process of being read, CPUs will try to guess the destination and attempt to execute ahead. When the memory value finally arrives, the CPU either discards or commits the speculative computation. Speculative logic is unfaithful in how it executes, can access the victim's memory and registers, and can perform operations with**

leverage hardware vulnerabilities to leak s
Attacks of the latter type include micro
exploiting cache timing [8, 30, 48, 52,
prediction history [1, 2], branch target buf
DRAM rows [56]. Software-based techni
used to mount fault attacks that alter phys
internal CPU values [65].

CacheBleed: A Timing Attack on OpenSSL Constant Time RSA

Yuval Yarom
The University of Adelaide and NICTA

Daniel Genkin
Technion and Tel Aviv University

Nadia Heninger
University of Pennsylvania

Overview

CacheBleed is a side-channel attack that exploits inform
Intel processors. By detecting cache-bank conflicts via m
recover information about victim processes running on th
recover both 2048-bit and 4096-bit RSA secret keys from
Bridge processors after observing only 16,000 secret-key
is despite the fact that OpenSSL's RSA implementation v
in order to protect against cache-based (and other) side-

Meltdown: Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2],
Werner Haas[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1],
Paul Kocher[5], Daniel Genkin[6,9], Yuval Yarom[7], Mike Hamburg[8]
[1]*Graz University of Technology,* [2]*Cyberus Technology GmbH,*
[3]*G-Data Advanced Analytics,* [4]*Google Project Zero,*
[5]*Independent (www.paulkocher.com),* [6]*University of Michigan,*
[7]*University of Adelaide & Data61,* [8]*Rambus, Cryptography Research Division*

amentally relies
ress ranges are
ected from user
own. Meltdown
cution on mod-
emory locations
. Out-of-order
nce feature and

sor bit of the processor that defines whether a memory page of the kernel can be accessed or not. The basic idea is that this bit can only be set when entering kernel code and it is cleared when switching to user processes. This hardware feature allows operating systems to map the kernel into the address space of every process and to have very efficient transitions from the user process to the kernel, e.g., for interrupt handling. Consequently, in practice, there is no change of the memory mapping when switching from a user process to the kernel

2015 IEEE Symposium on Security and Privacy

Last-Level Cache Side-Channel Attacks are Practical

Fangfei Liu*†, Yuval Yarom*‡§, Qian Ge§¶, Gernot Heiser§¶, Ruby B. Lee†
* Equal contribution joint first authors.
† Department of Electrical Engineering, Princeton University
Email: {fangfeil,rblee}@princeton.edu
‡ School of Computer Science, The University of Adelaide
Email: yval@cs.adelaide.edu.au
§ NICTA
Email: {qian.ge,gernot}@nicta.com.au
¶ UNSW Australia

***Abstract*—We present an effective implementation of**

Goal: eliminate bugs across the stack

Systematic approach to rule out a large class of bugs in software and hardware:

- Correctness bugs
- Security bugs
- Timing side-channel leakage

Approach: formal verification

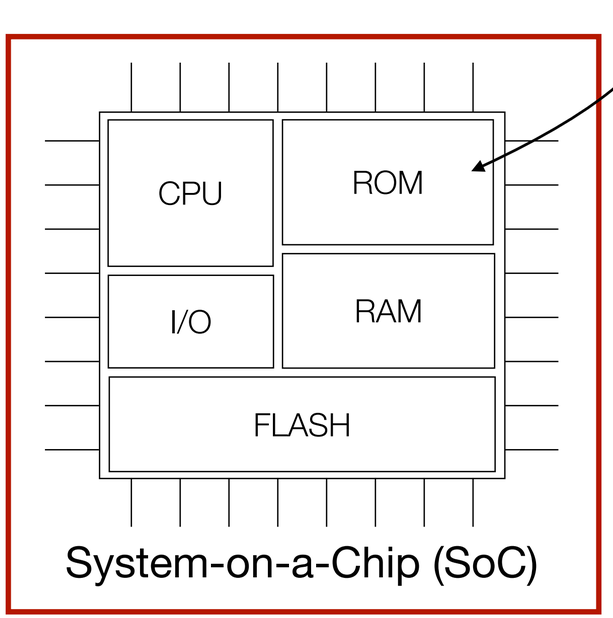
```
818   finv(zinv, pz);
819   fmul(res, px, zinv);
820   from_mont(res, res);
821 }
822
823 static void point_double(uint64_t *res, uint64_t *p)
824 {
825   uint64_t tmp[20U] = { 0U };
826   uint64_t *x = p;
827   uint64_t *z = p + 8U;
828   uint64_t *x3 = res;
829   uint64_t *y3 = res + 4U;
830   uint64_t *z3 = res + 8U;
831   uint64_t *t0 = tmp;
832   uint64_t *t1 = tmp + 4U;
833   uint64_t *t2 = tmp + 8U;
834   uint64_t *t3 = tmp + 12U;
835   uint64_t *t4 = tmp + 16U;
836   uint64_t *x1 = p;
837   uint64_t *y = p + 4U;
838   uint64_t *z1 = p + 8U;
839   fsqr(t0, x1);
840   fsqr(t1, y);
841   fsqr(t2, z1);
842   fmul(t3, x1, y);
843   fadd(t3, t3, t3);
```

```
h = hash(msg)
k = rand()
R = k * G
r = R.x
s = k^-1 * (h + p * r) mod n
return (r, s)
```

Mathematical specification

~ 100 LoC

$= IPR$



Entire hardware/software system

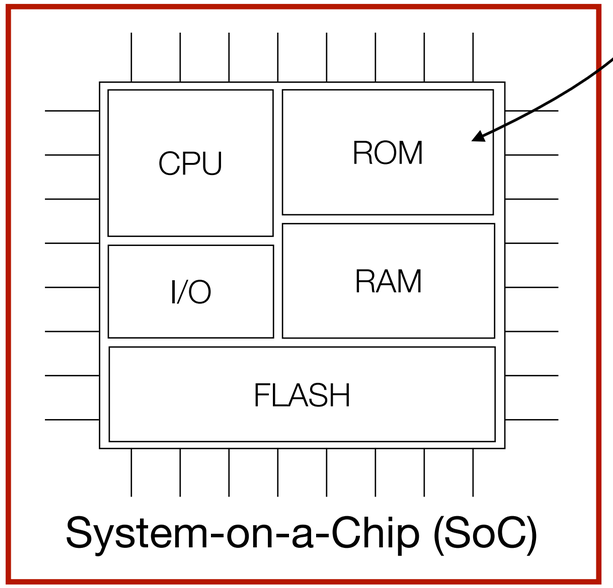
~ 10,000 LoC

Approach: formal verification

Information-Preserving Refinement:
captures correctness, security, and non-leakage

```
h = hash(msg)
k = rand()
R = k * G
r = R.x
s = k^-1 * (h + p * r) mod n
return (r, s)
```

$\overset{IPR}{=}$



Mathematical specification
~ 100 LoC

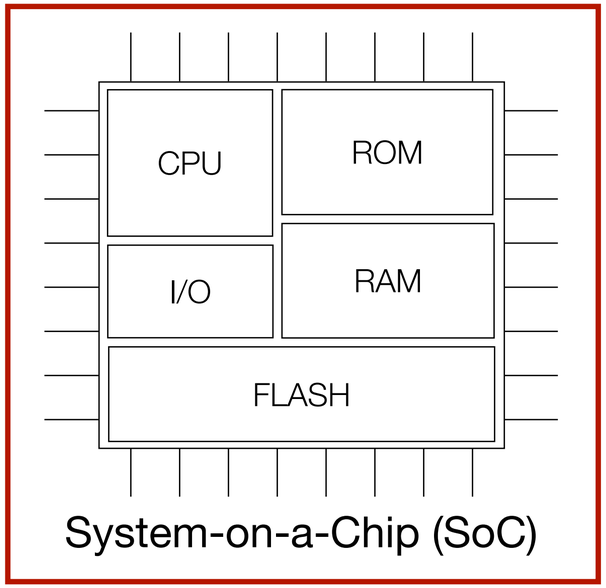
Entire hardware/software system
~ 10,000 LoC

```
818   finv(zinv, pz);
819   fmul(res, px, zinv);
820   from_mont(res, res);
821 }
822
823 static void point_double(uint64_t *res, uint64_t *p)
824 {
825   uint64_t tmp[20U] = { 0U };
826   uint64_t *x = p;
827   uint64_t *z = p + 8U;
828   uint64_t *x3 = res;
829   uint64_t *y3 = res + 4U;
830   uint64_t *z3 = res + 8U;
831   uint64_t *t0 = tmp;
832   uint64_t *t1 = tmp + 4U;
833   uint64_t *t2 = tmp + 8U;
834   uint64_t *t3 = tmp + 12U;
835   uint64_t *t4 = tmp + 16U;
836   uint64_t *x1 = p;
837   uint64_t *y = p + 4U;
838   uint64_t *z1 = p + 8U;
839   fsqr(t0, x1);
840   fsqr(t1, y);
841   fsqr(t2, z1);
842   fmul(t3, x1, y);
843   fadd(t3, t3, t3);
```

Approach: formal verification

```
h = hash(msg)
k = rand()
R = k * G
r = R.x
s = k^-1 * (h + p * r) mod n
return (r, s)
```

$\models_{IPR}$



Mathematical specification

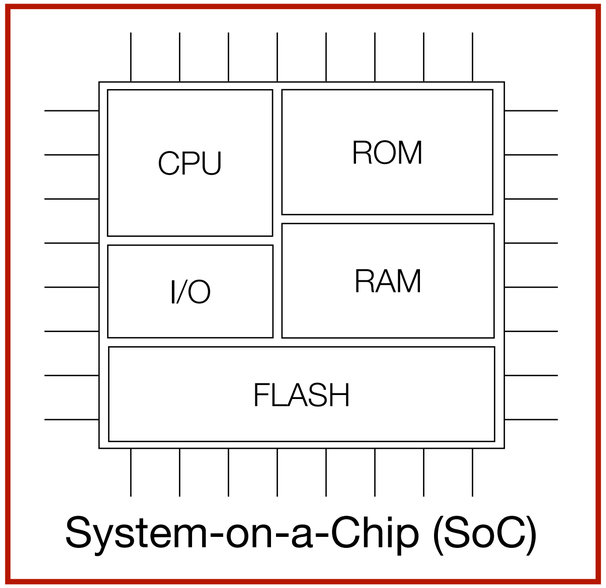
Entire hardware/software system

Approach: formal verification

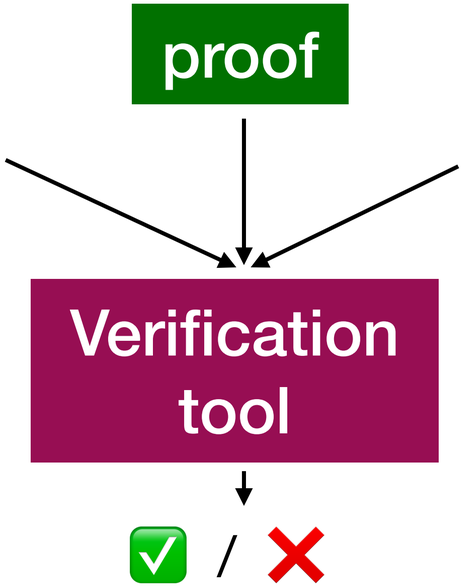
```
h = hash(msg)
k = rand()
R = k * G
r = R.x
s = k^-1 * (h + p * r) mod n
return (r, s)
```

Mathematical specification

$= IPR$



Entire hardware/software system



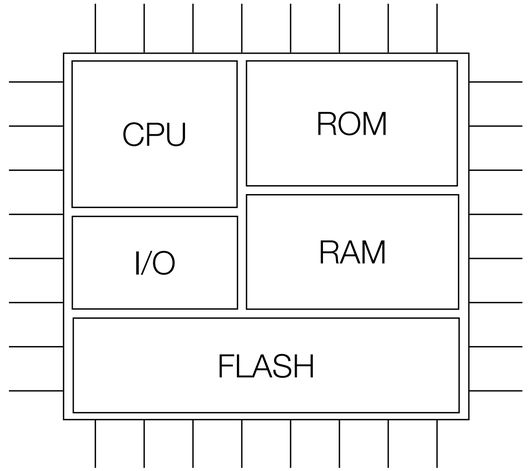
Challenge 1: gap between specification and implementation

Huge gap between mathematical specification and circuit-level implementation

Software: optimized code for crypto

Hardware: pipelined processor

```
h = hash(msg)
k = rand()
R = k * G
r = R.x
s = k^-1 * (h + p * r) mod n
return (r, s)
```



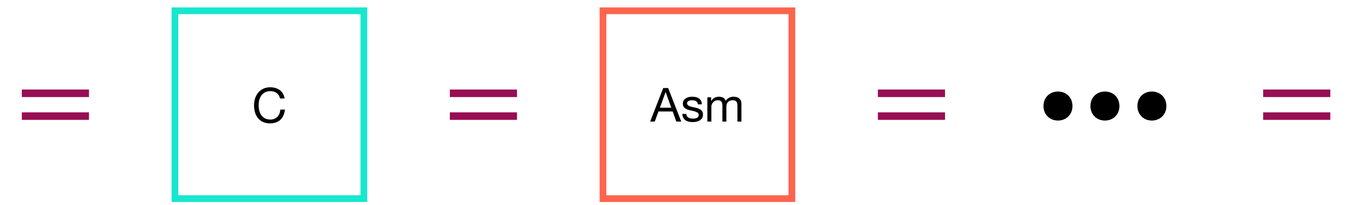
System-on-a-Chip (SoC)

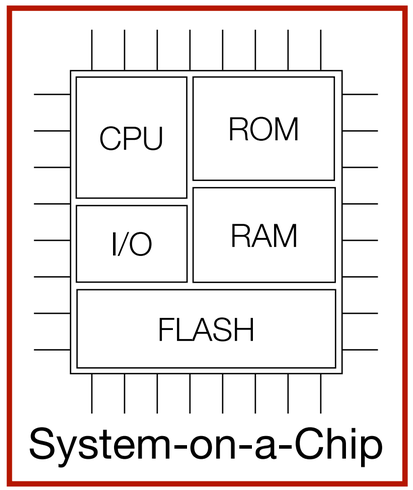
Solution: proof modularity using transitive IPR

Break down the proof into more manageable pieces

- Formalized and proved transitivity of IPR
- Separate reasoning about software, compilation, and hardware

```
h = hash(msg)
k = rand()
R = k * G
r = R.x
s = k^-1 * (h + p * r) mod n
return (r, s)
```





Challenge 2: different ways of thinking about equivalences

Prior work has looked at connecting specs to C, C to Asm, ...

Different ways of thinking about equivalences, that capture different properties

Pre/post-conditions

Refinement

Compiler correctness

Noninterference

Constant-time

Solution: IPR as a common framework

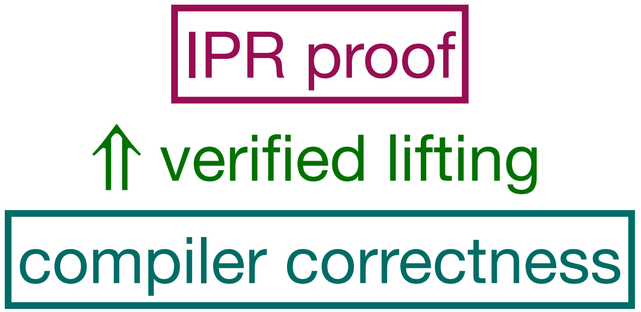
IPR as the consistent way to think about equivalences across levels of abstraction

Lift existing proofs to IPR where possible

Software, compilers

Introduce new techniques otherwise

Hardware



Contributions

Transitive IPR: scaling proofs of IPR with modularity

Proof techniques for IPR: proving IPR across the software/hardware stack

Parfait verification framework: implements these techniques

Application of Parfait to Hardware Security Modules (HSMs):

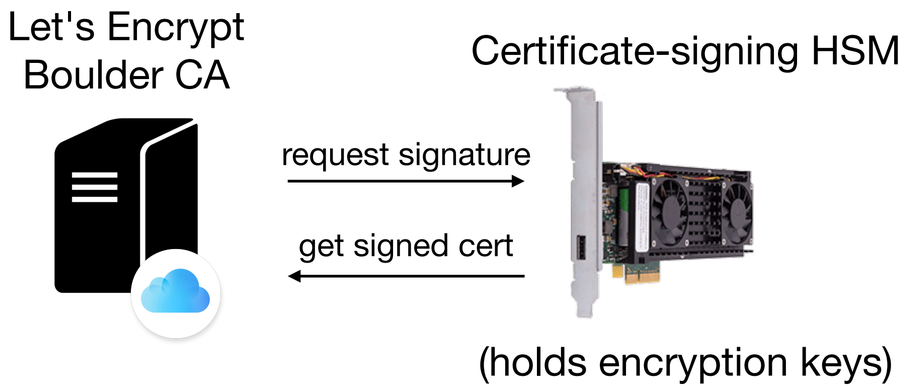
Including an ECDSA-signing HSM (2,300 LoC and 13,500 lines of Verilog) verified against a 40 LoC spec (on top of specs from HACL*)

anish.io/parfait

Context: hardware security modules (HSMs)

Factor out core security-critical functionality to a separate device

Billions of deployed HSMs: U2F token, iPhone Secure Enclave, PKCS#11 HSM, Apple Cloud Key Vault, ...



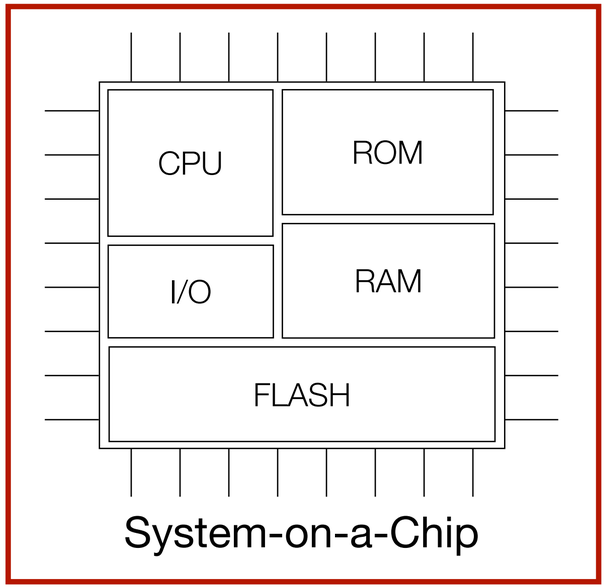
Information-Preserving Refinement (IPR) [Knox, OSDI'22]

```
var prf_key, prf_counter, private_key

def initialize(new_prf_key, new_private_key):
    prf_key = new_prf_key
    prf_counter = 0
    private_key = new_private_key

def sign(message):
    if prf_counter == 2^64 - 1:
        return Error
    nonce = hmac_sha256(prf_key, prf_counter)
    prf_counter += 1
    return ecdsa_p256(message, private_key, nonce)
```

ECDSA signing HSM spec

~ 40 LoC

$\Rightarrow_{IPR}$



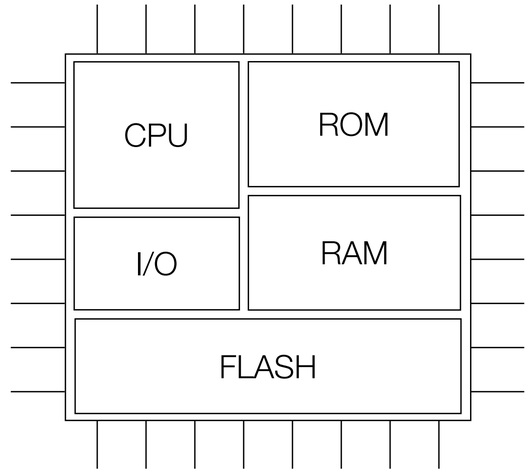
Entire hardware/software system

~ 15,000 LoC

Information-Preserving Refinement (IPR) [Knox, OSDI'22]

```
var prf_key, prf_counter, private_key

def initialize(new_prf_key, new_private_key):
    prf_key = new_prf_key
    prf_counter = 0
    private_key = new_private_key

def sign(message):
    if prf_counter == 2^64 - 1:
        return Error
    nonce = hmac_sha256(prf_key, prf_counter)
    prf_counter += 1
    return ecdsa_p256(message, private_key, nonce)
```

$\overset{?}{=}_{IPR}$



System-on-a-Chip

ECDSA signing HSM spec

Entire hardware/software system

~ 15,000 LoC

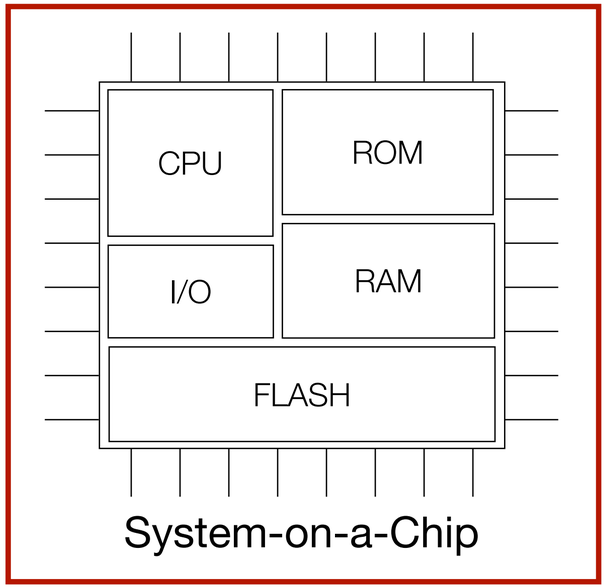
Specification

interaction model: whole-command state machine, only observables are function calls and return values (no notion of timing)

Information-Preserving Refinement (IPR) [Knox, OSDI'22]

```
var prf_key, prf_counter, private_key

def initialize(new_prf_key, new_private_key):
    prf_key = new_prf_key
    prf_counter = 0
    private_key = new_private_key

def sign(message):
    if prf_counter == 2^64 - 1:
        return Error
    nonce = hmac_sha256(prf_key, prf_counter)
    prf_counter += 1
    return ecdsa_p256(message, private_key, nonce)
```

$\overset{?}{=}_{IPR}$



ECDSA signing HSM spec

~ 40 LoC

Entire hardware/software system

Implementation

interaction model: cycle-precise digital wire-level I/O

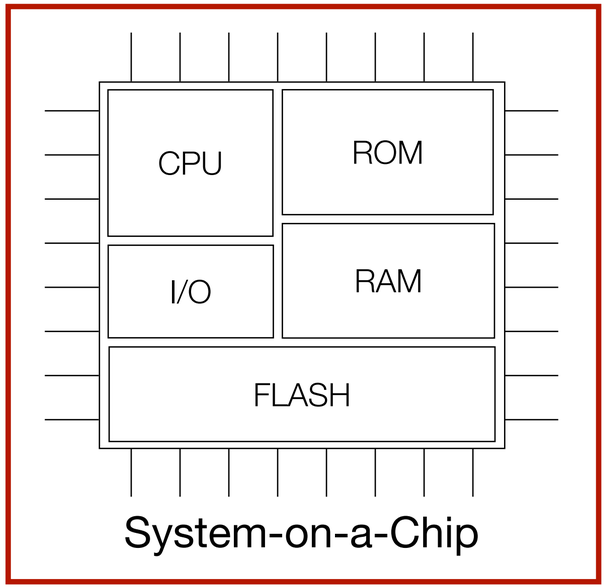
Information-Preserving Refinement (IPR) [Knox, OSDI'22]

```
var prf_key, prf_counter, private_key

def initialize(new_prf_key, new_private_key):
    prf_key = new_prf_key
    prf_counter = 0
    private_key = new_private_key

def sign(message):
    if prf_counter == 2^64 - 1:
        return Error
    nonce = hmac_sha256(prf_key, prf_counter)
    prf_counter += 1
    return ecdsa_p256(message, private_key, nonce)
```

ECDSA signing HSM spec

~ 40 LoC

$\Rightarrow_{IPR}$

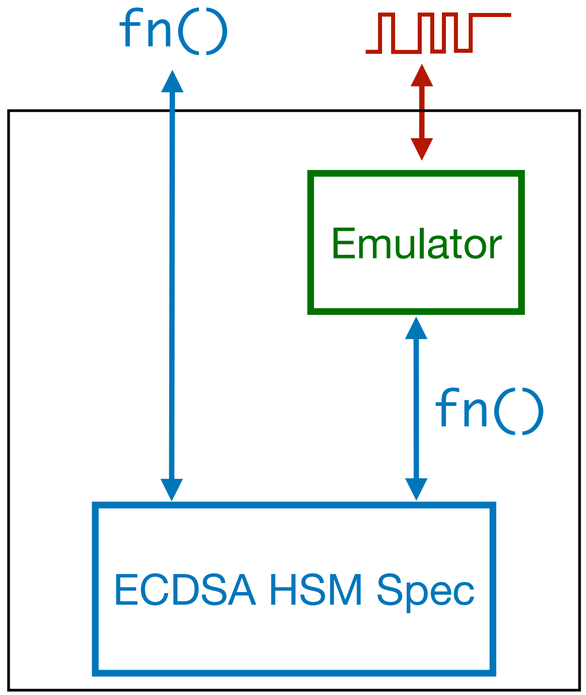


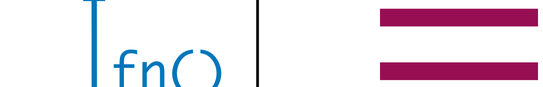
Entire hardware/software system

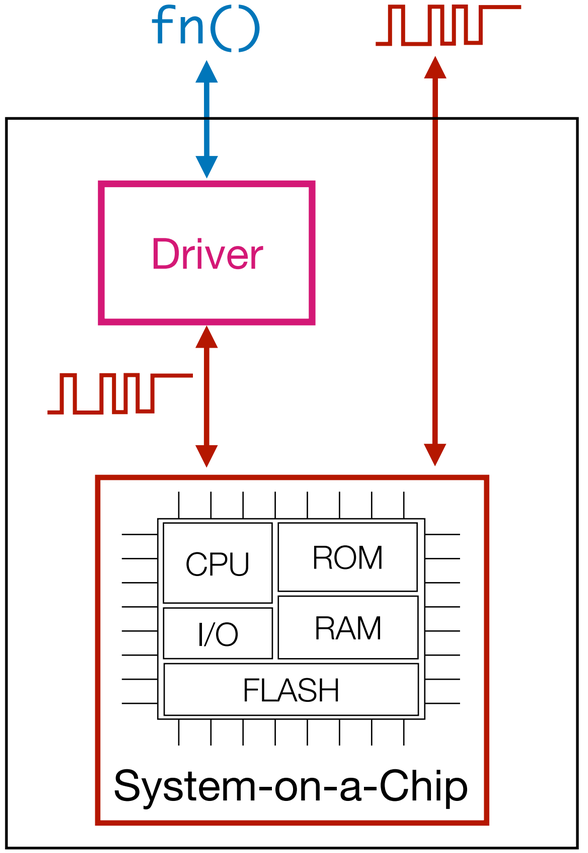
~ 15,000 LoC

Information-Preserving Refinement (IPR) [Knox, OSDI'22]

Ideal World



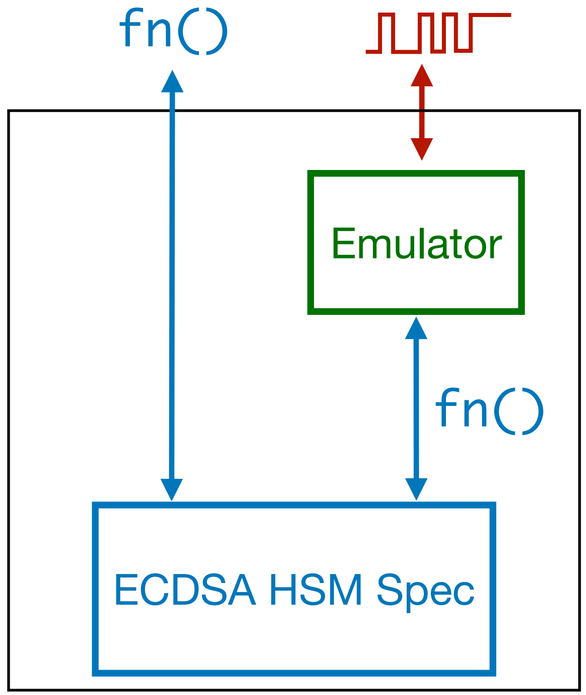




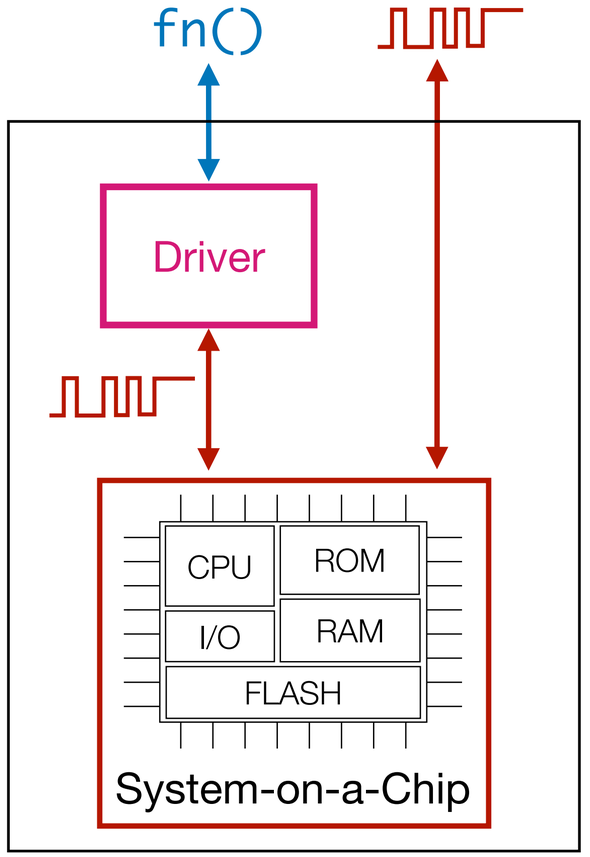
Real World

Information-Preserving Refinement (IPR) [Knox, OSDI'22]

Ideal World



=



Real World

Circuit implements spec, and it doesn't leak any additional information through its cycle-precise wire-level behavior

Captures correctness and security

Information-Preserving Refinement (IPR) [Knox, OSDI'22]

Key intuition of how IPR captures timing:

(1) model spec as a whole-command state machine,
(2) model implementation at the cycle-precise level, and
(3) show that we ***can exactly reproduce the implementation's timing behavior given access only to the spec***

HSM structure

System software
(I/O and persistence)

Written by platform developer

Core application logic (timing-sensitive application code)

Written by application developer

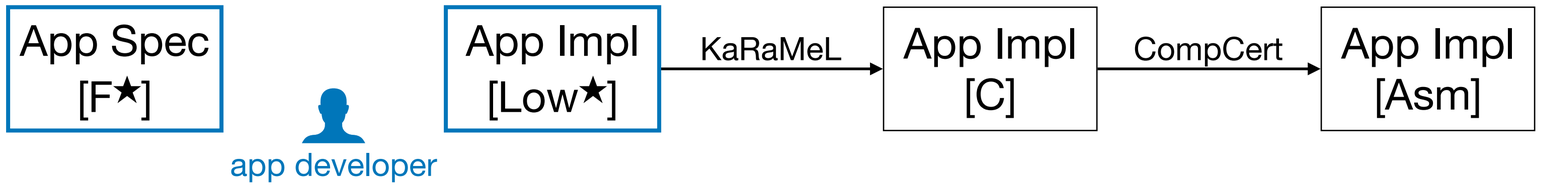
```
uint8_t state[STATE_SIZE];
uint8_t cmd[COMMAND_SIZE];
uint8_t resp[RESPONSE_SIZE];

void main() {
  while (1) {
    read_command(&cmd); // from I/O interface
    load_state(&state); // from persistent memory
    handle(&state, &cmd, &resp); // core computation
    store_state(&state); // to persistent memory, atomic
    write_response(&resp); // to I/O interface
  }
}
```

Parfait developer workflow: implementation

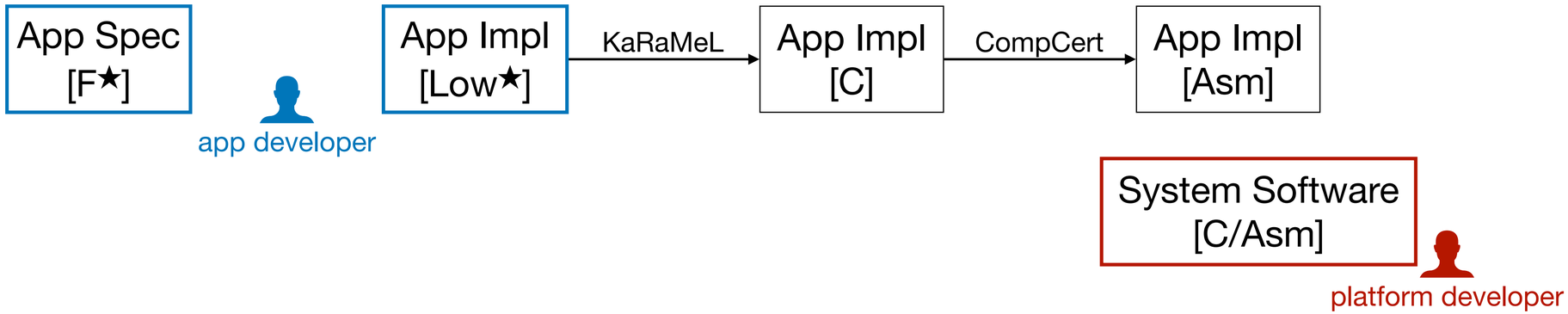
Parfait developer workflow: implementation

Parfait developer workflow: implementation

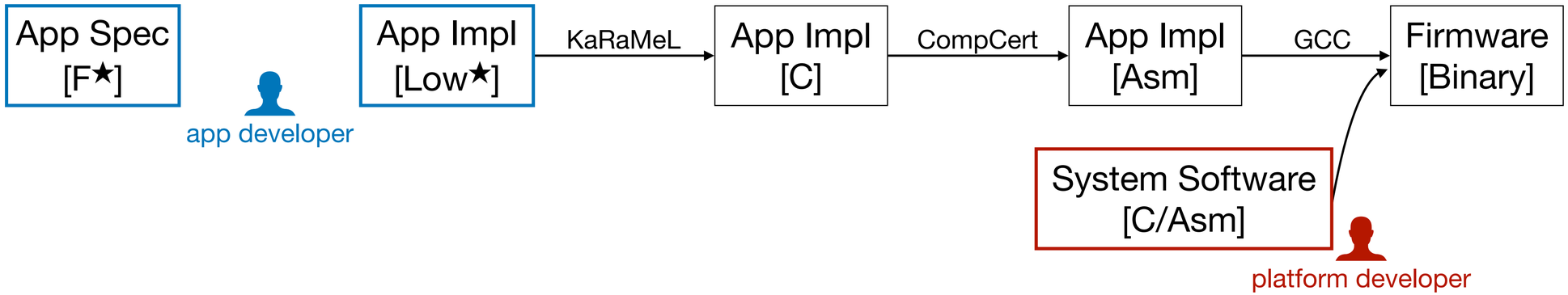
Parfait developer workflow: implementation



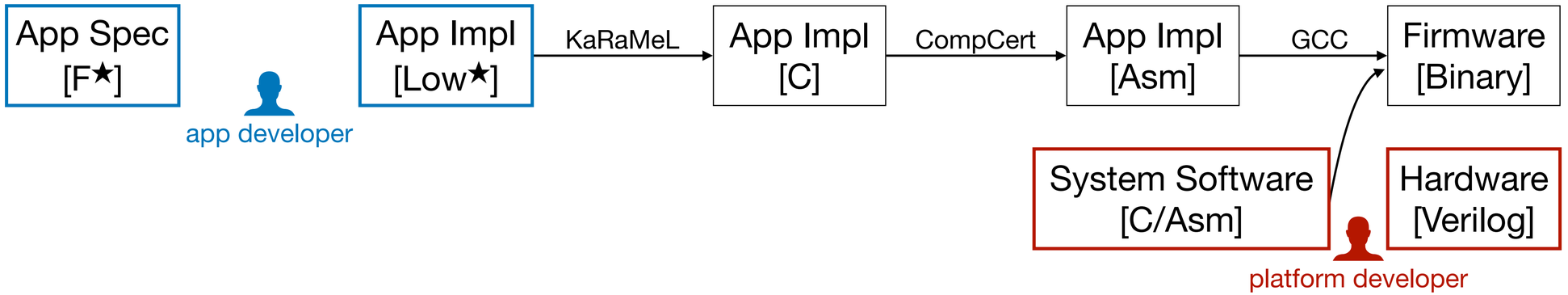
Parfait developer workflow: implementation



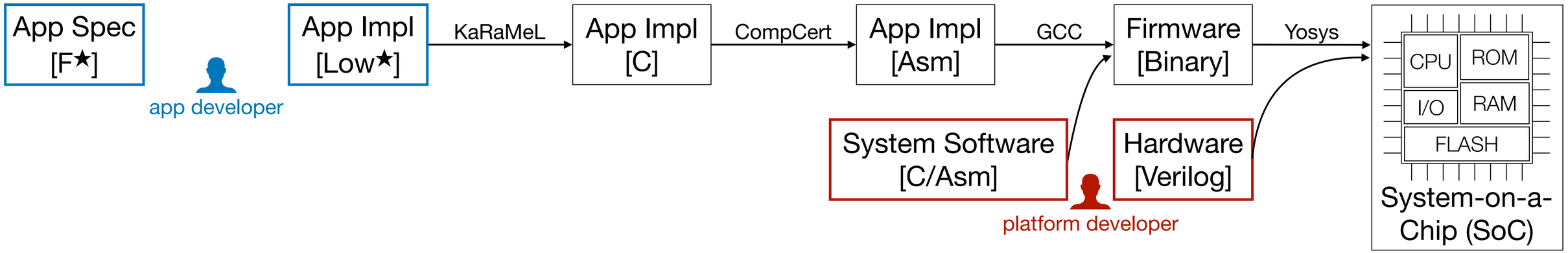
Parfait developer workflow: implementation



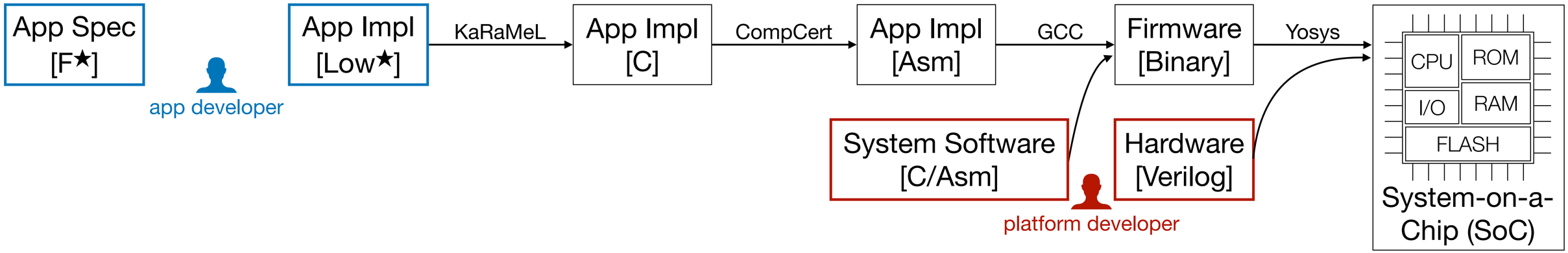
Parfait developer workflow: implementation



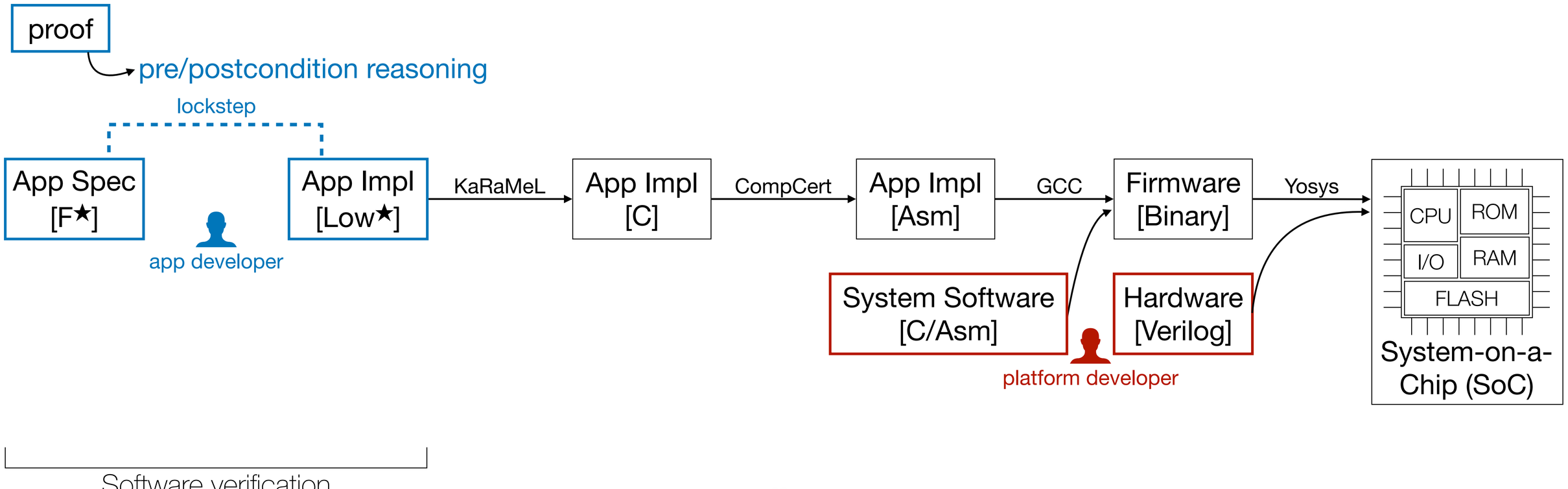
Parfait developer workflow: implementation



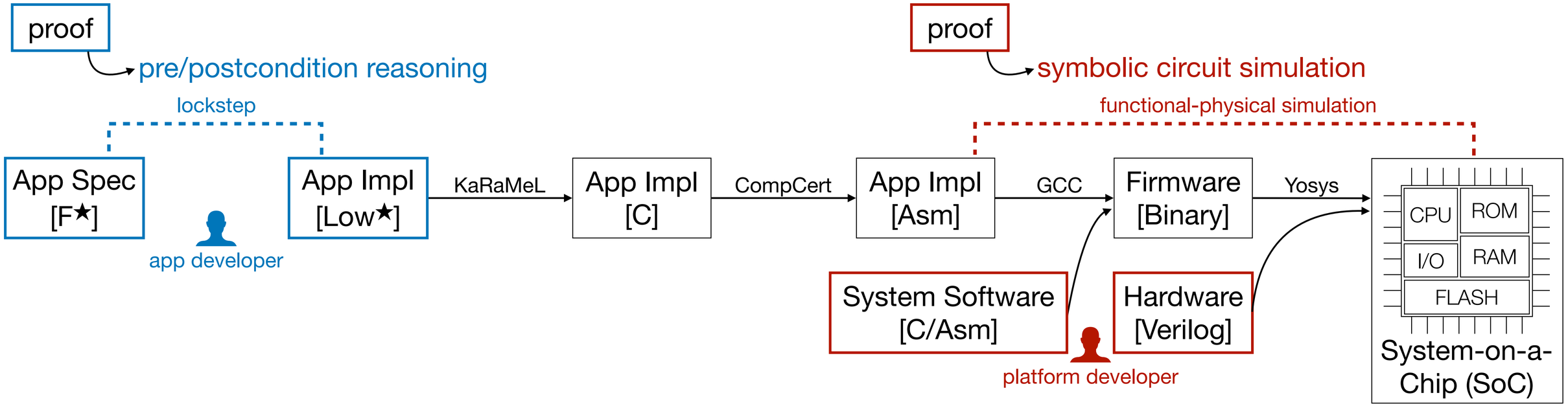
Parfait developer workflow: proofs



Parfait developer workflow: proofs



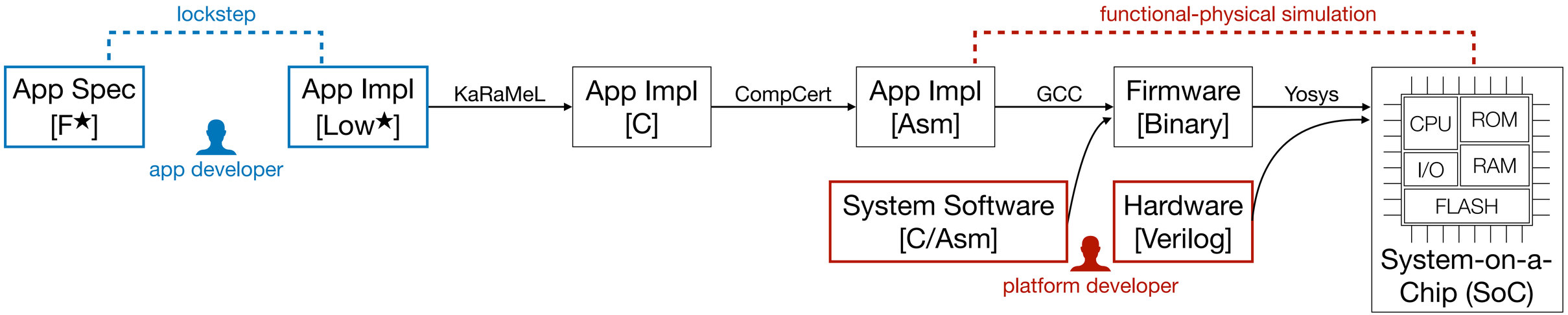
Parfait developer workflow: proofs



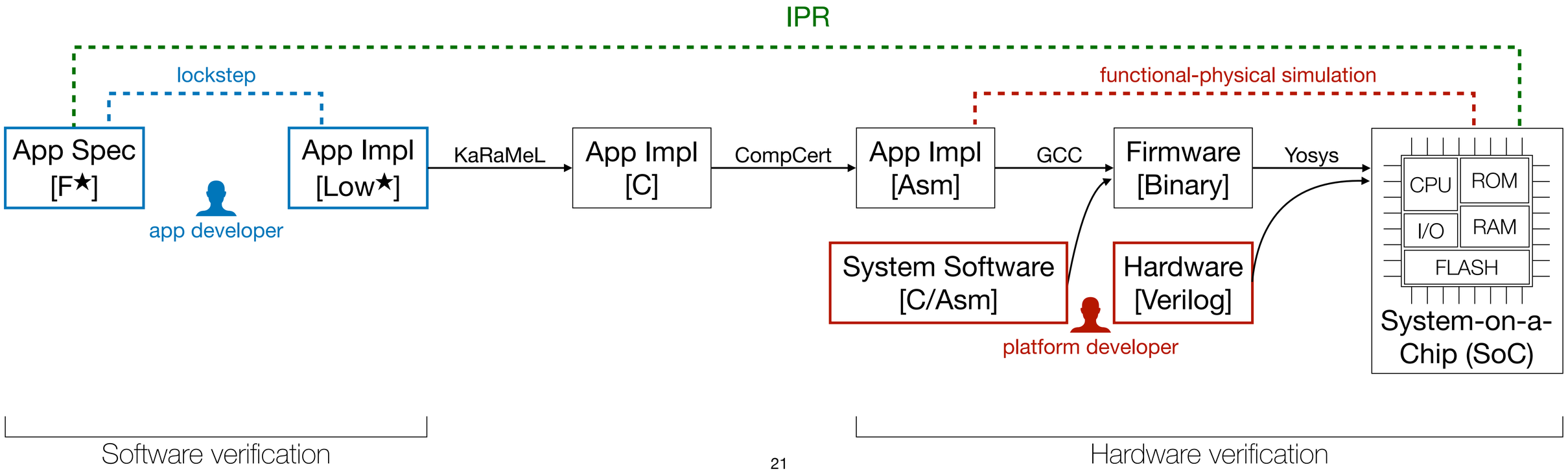
Software verification

Hardware verification

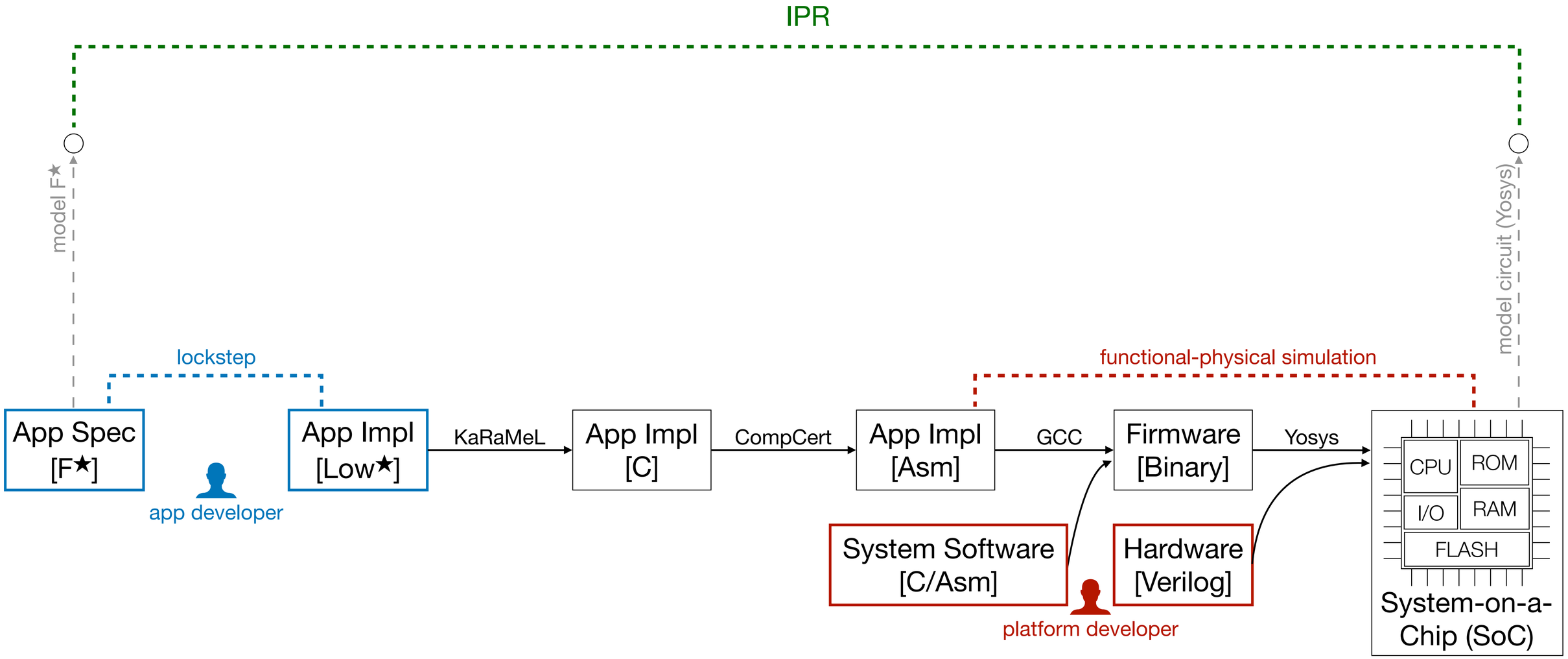
Parfait developer workflow: proofs



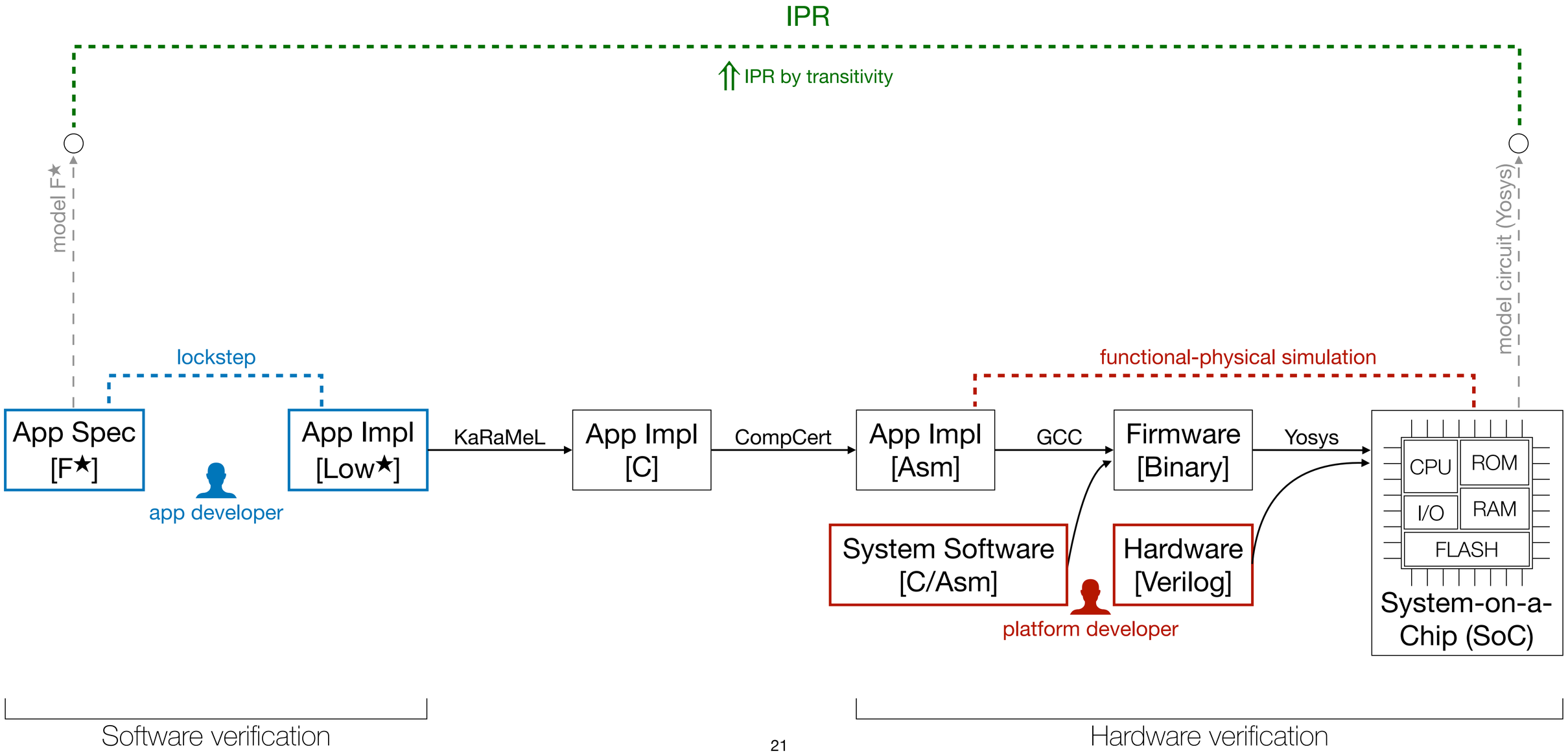
Parfait proof approach: modular verification with transitive IPR



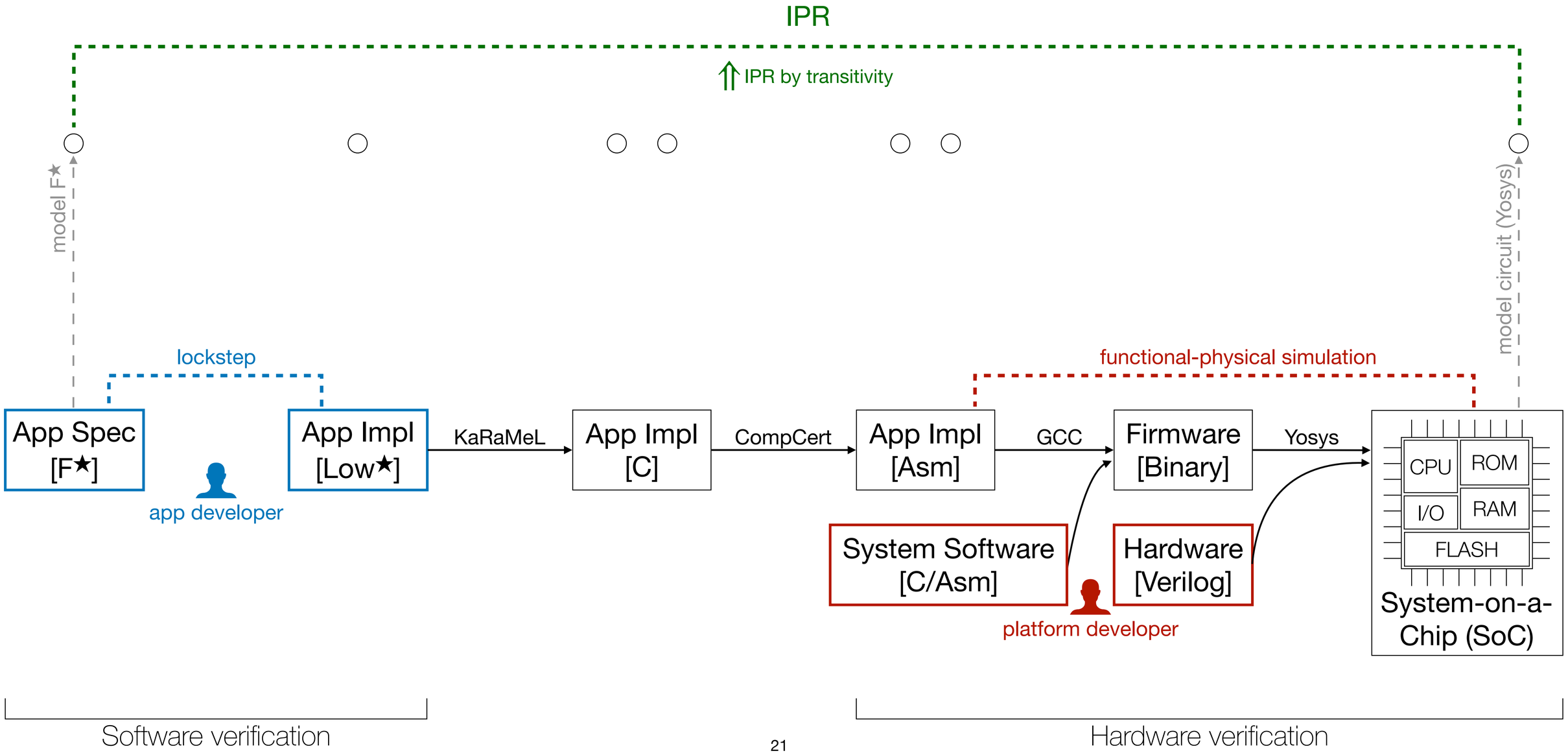
Parfait proof approach: modular verification with transitive IPR



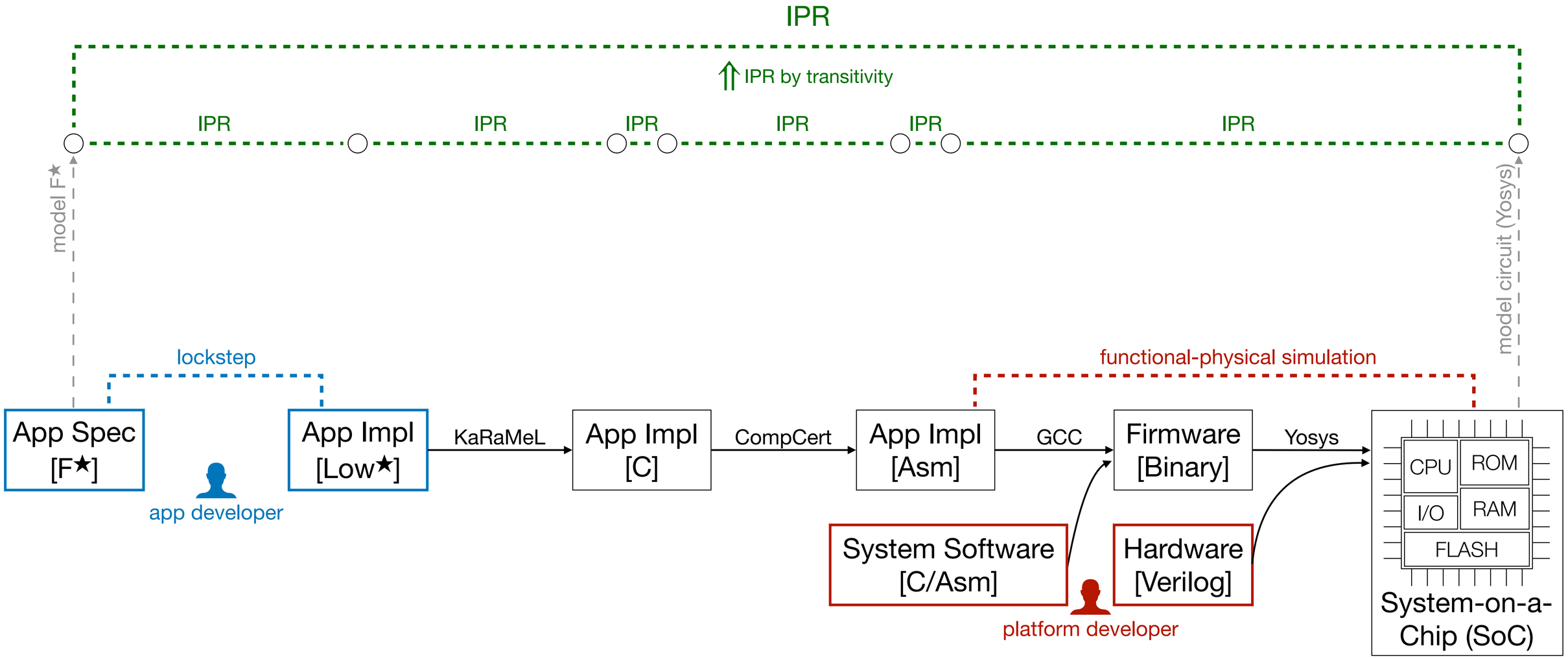
Parfait proof approach: modular verification with transitive IPR



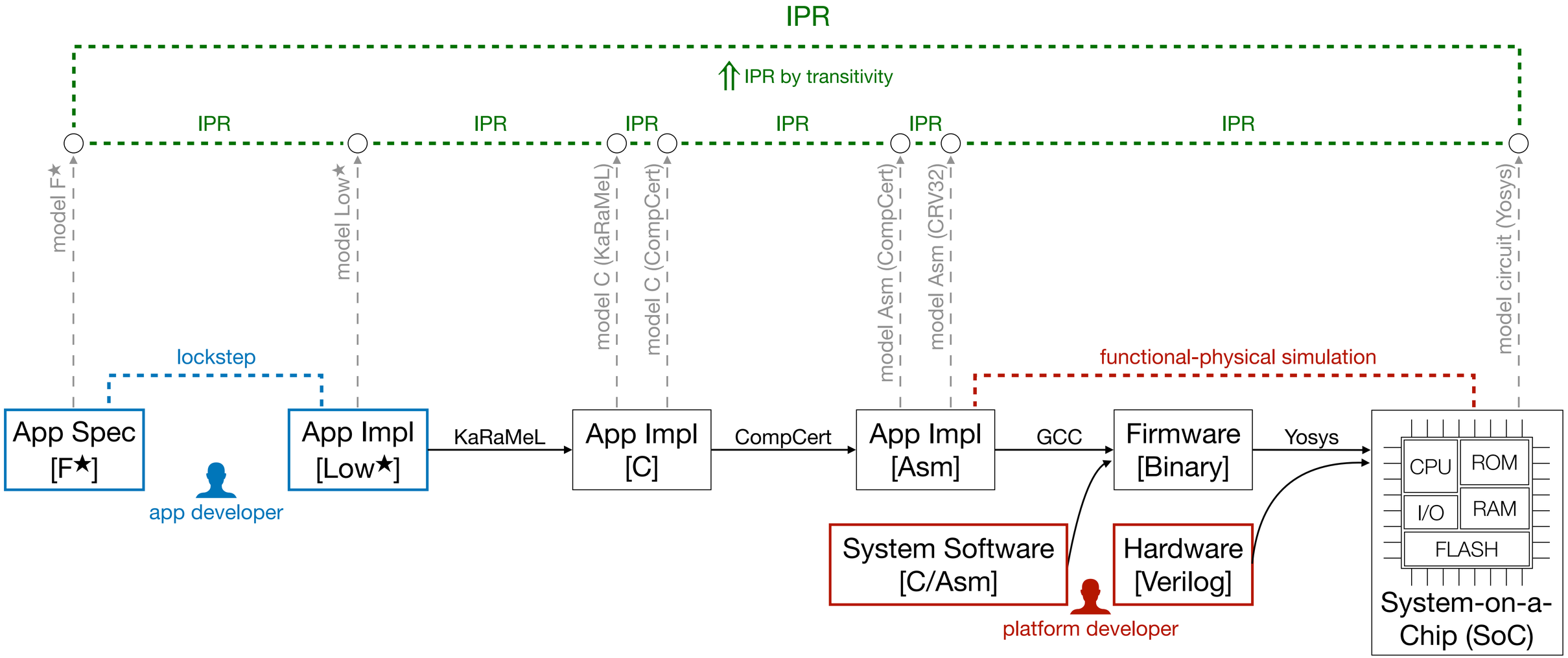
Parfait proof approach: modular verification with transitive IPR



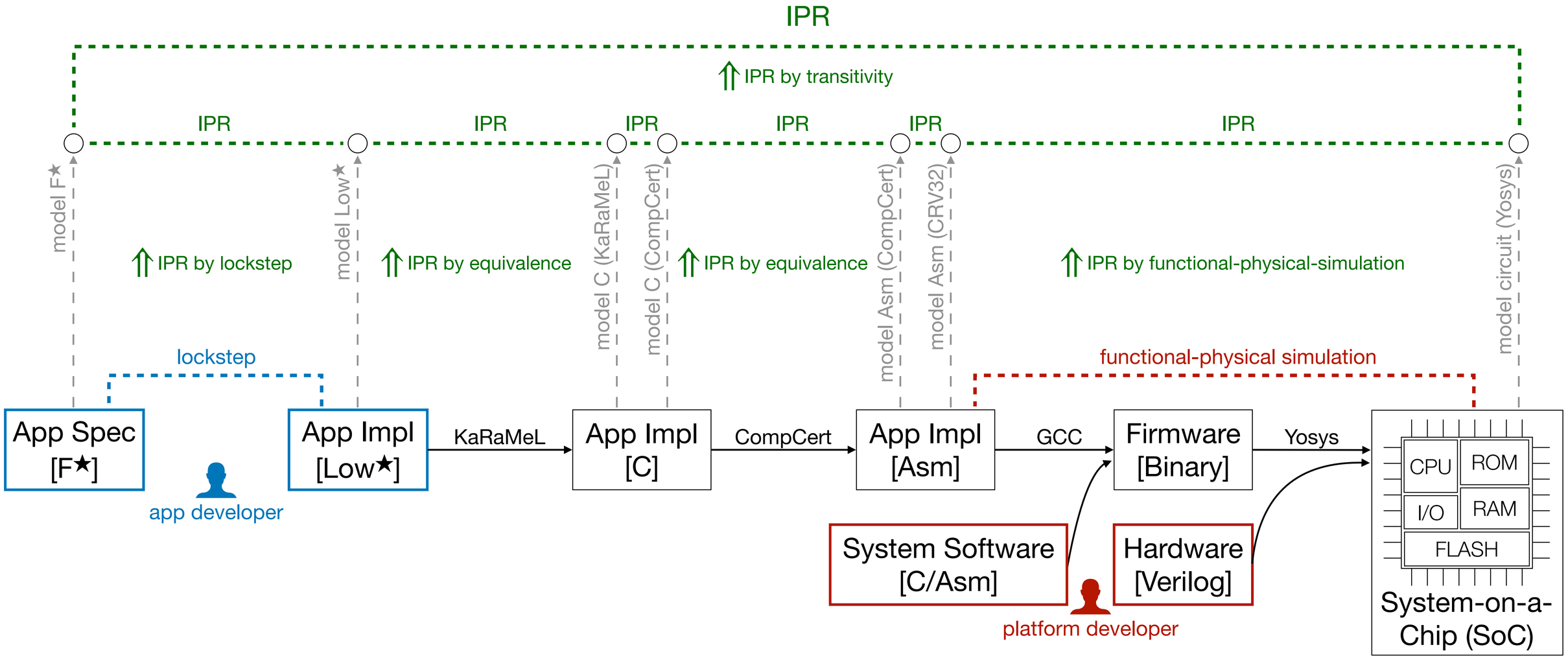
Parfait proof approach: modular verification with transitive IPR



Parfait proof approach: modular verification with transitive IPR



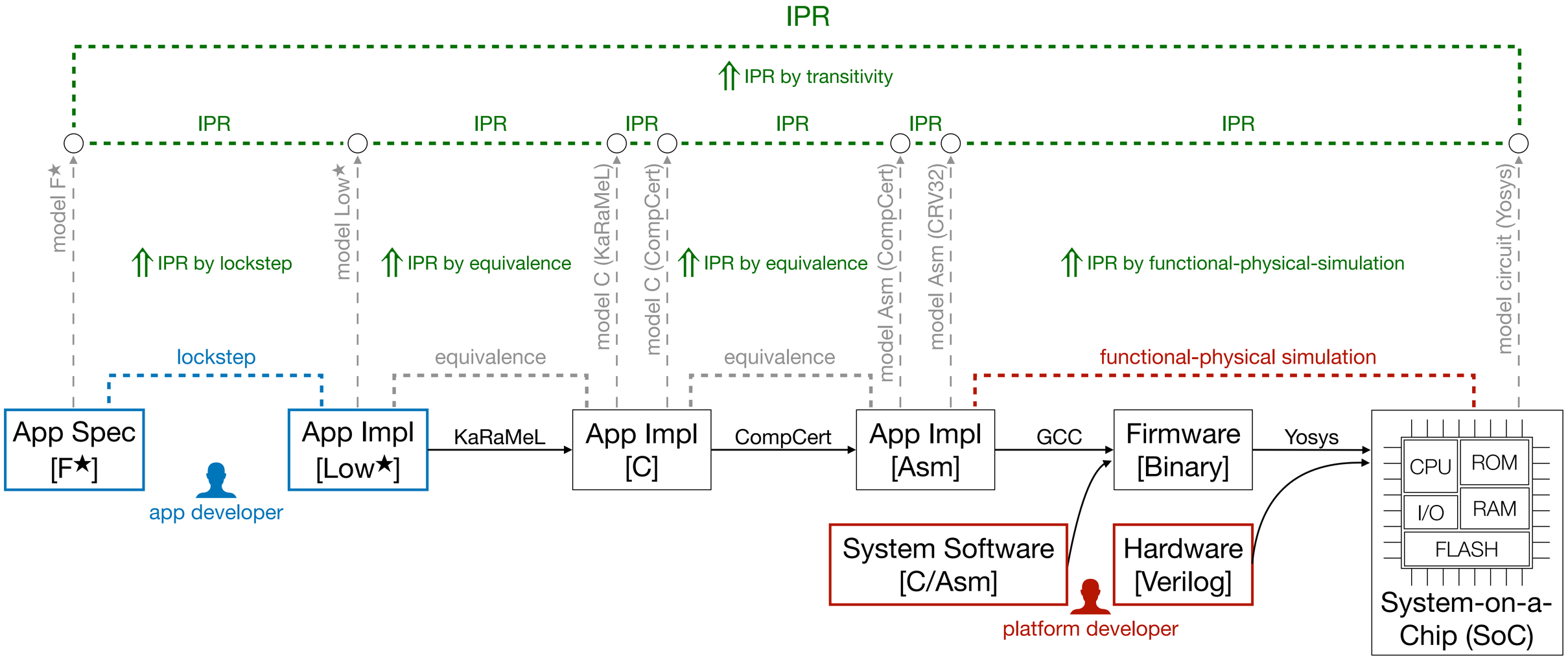
Parfait proof approach: modular verification with transitive IPR



Software verification

Hardware verification

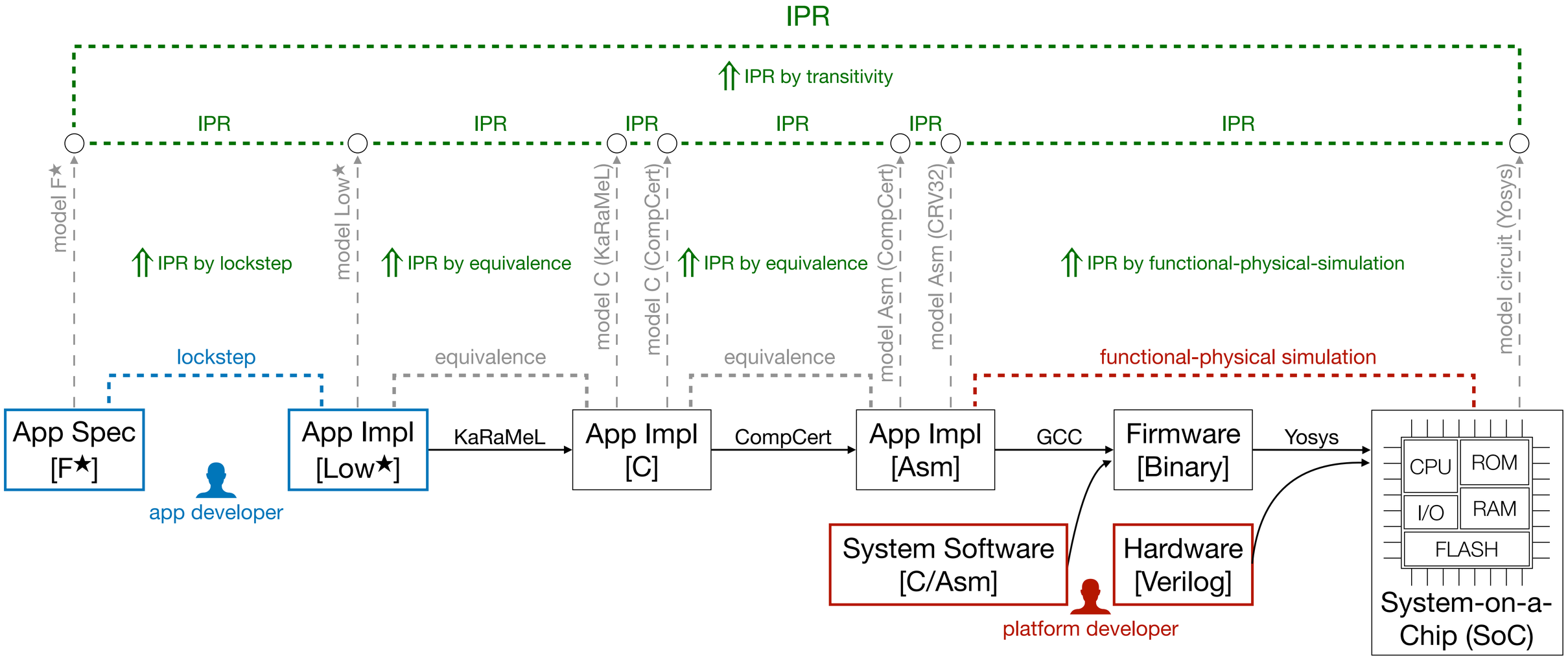
Parfait proof approach: modular verification with transitive IPR



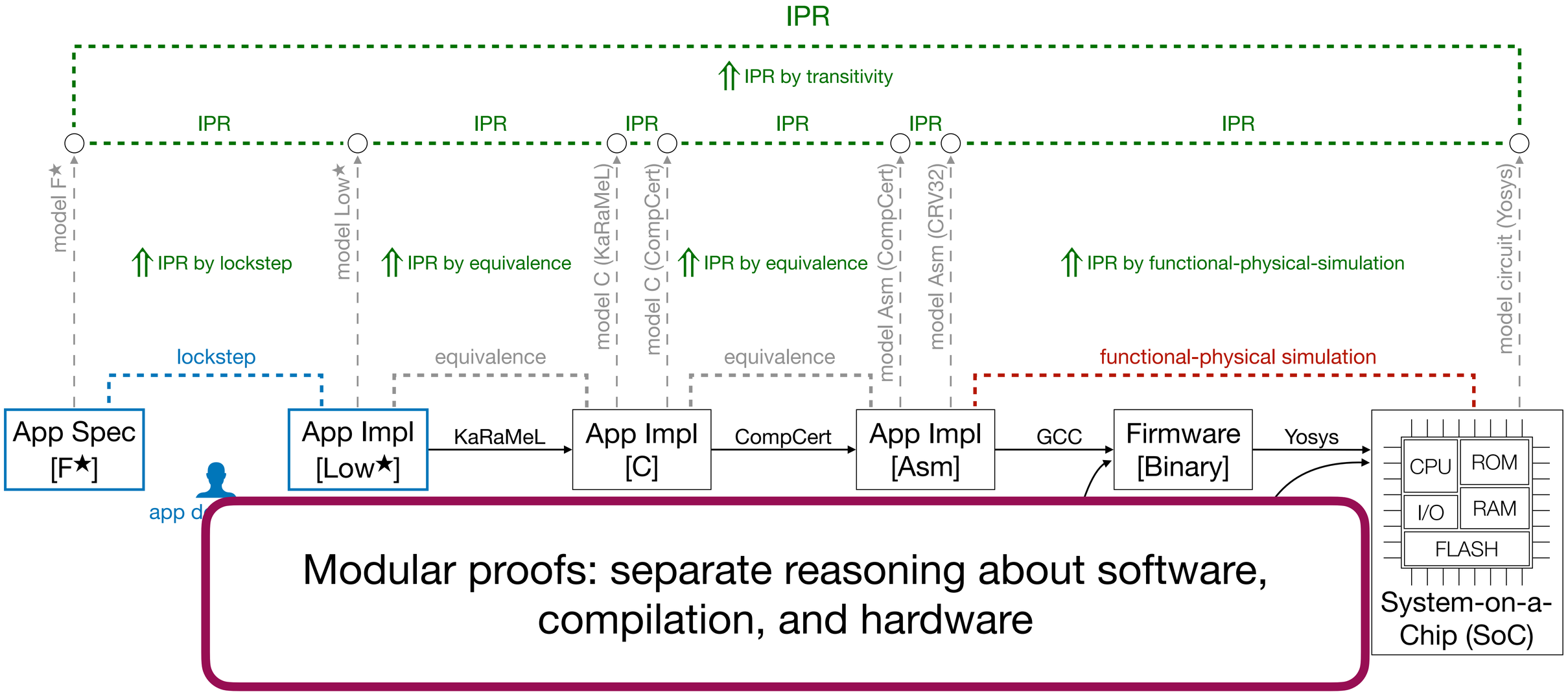
Software verification

Hardware verification

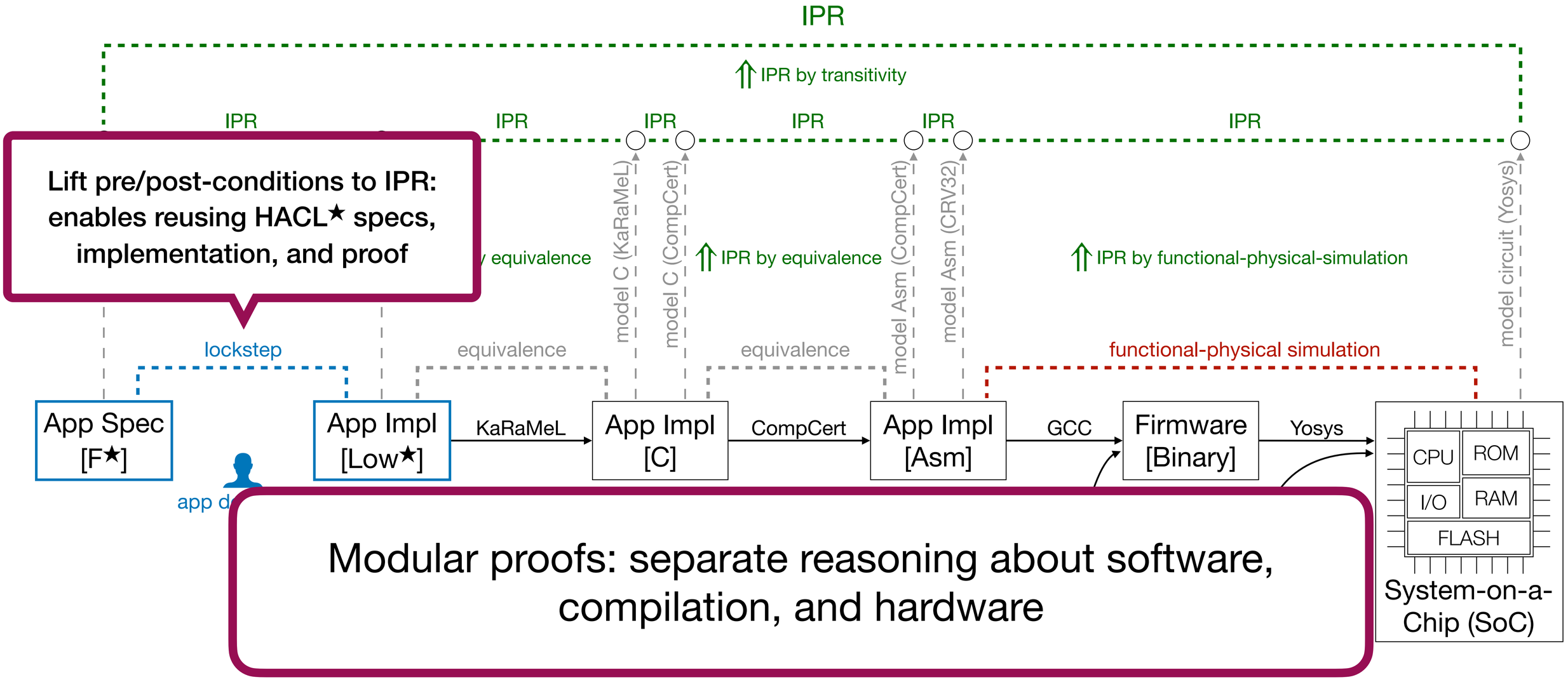
Benefits of the Parfait approach



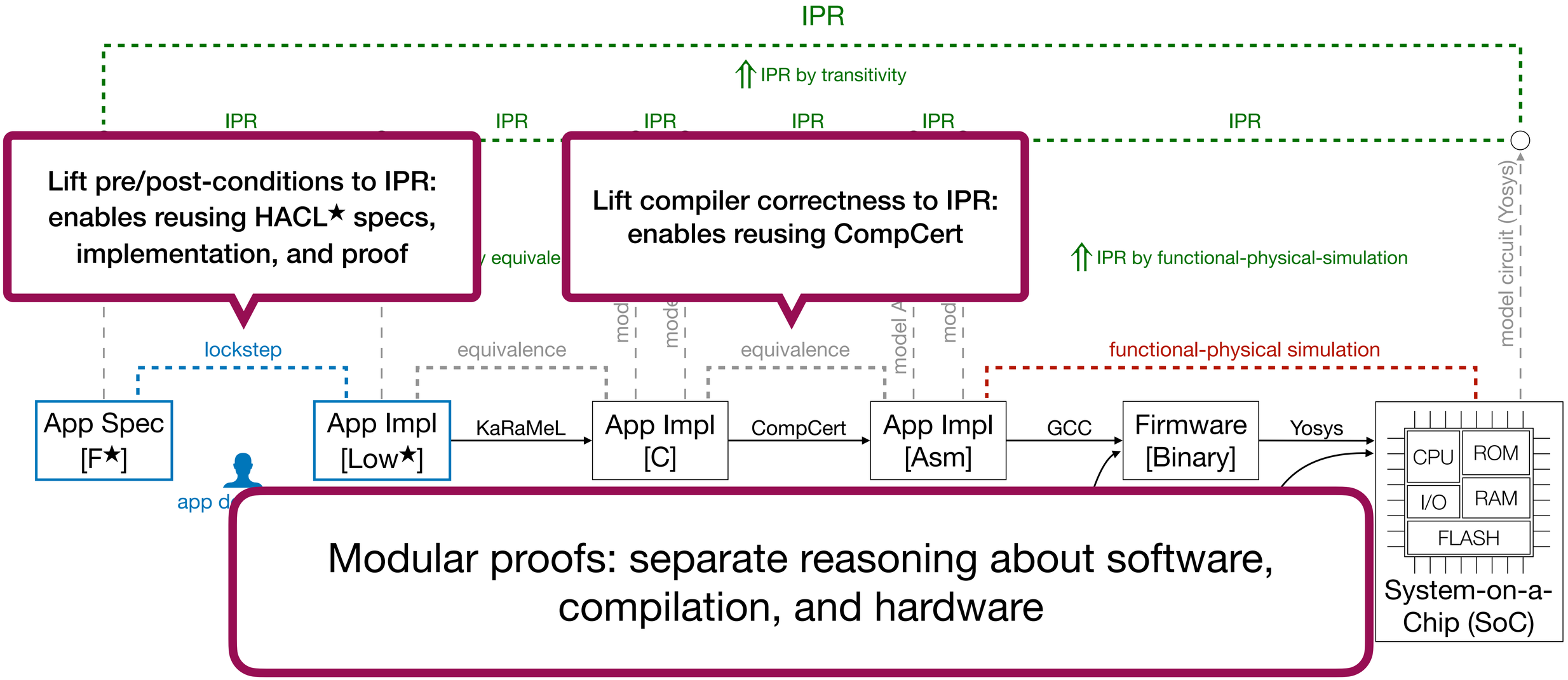
Benefits of the Parfait approach



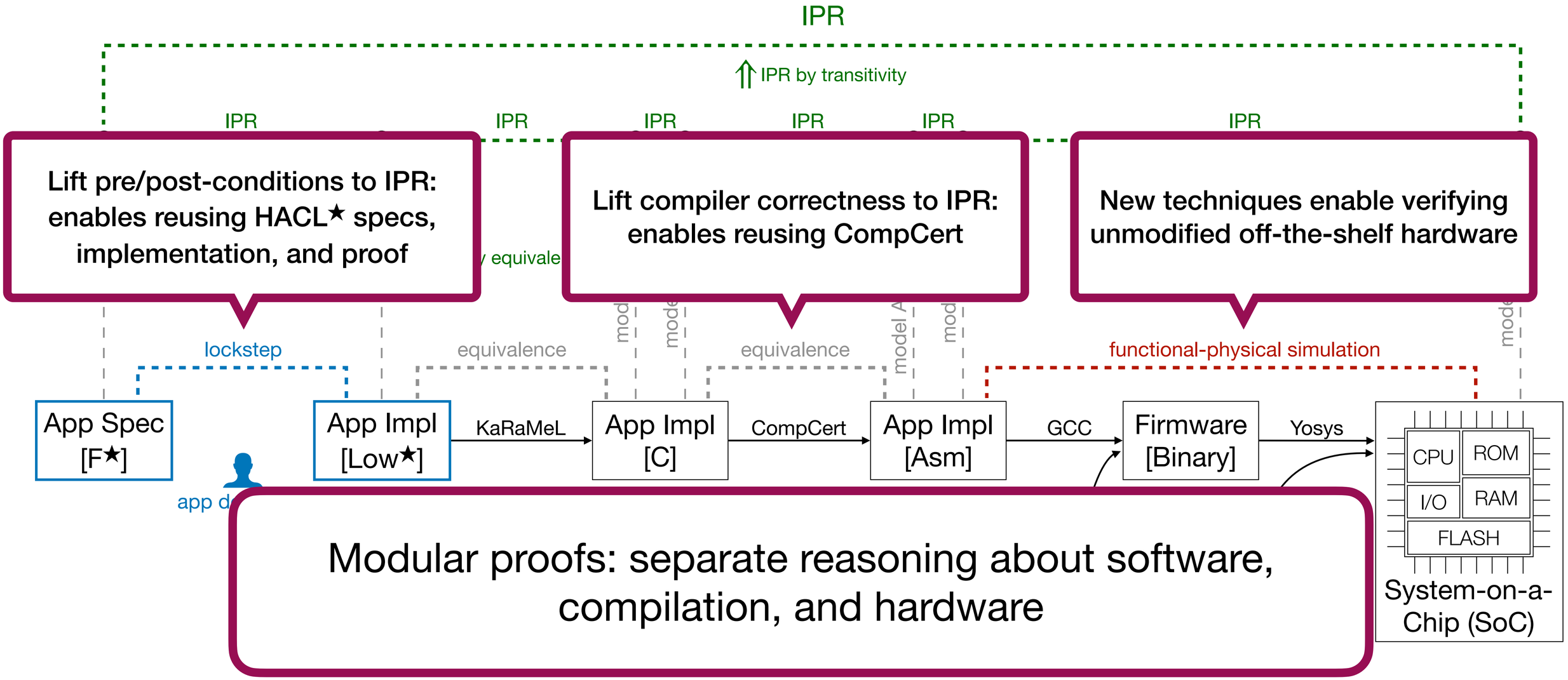
Benefits of the Parfait approach



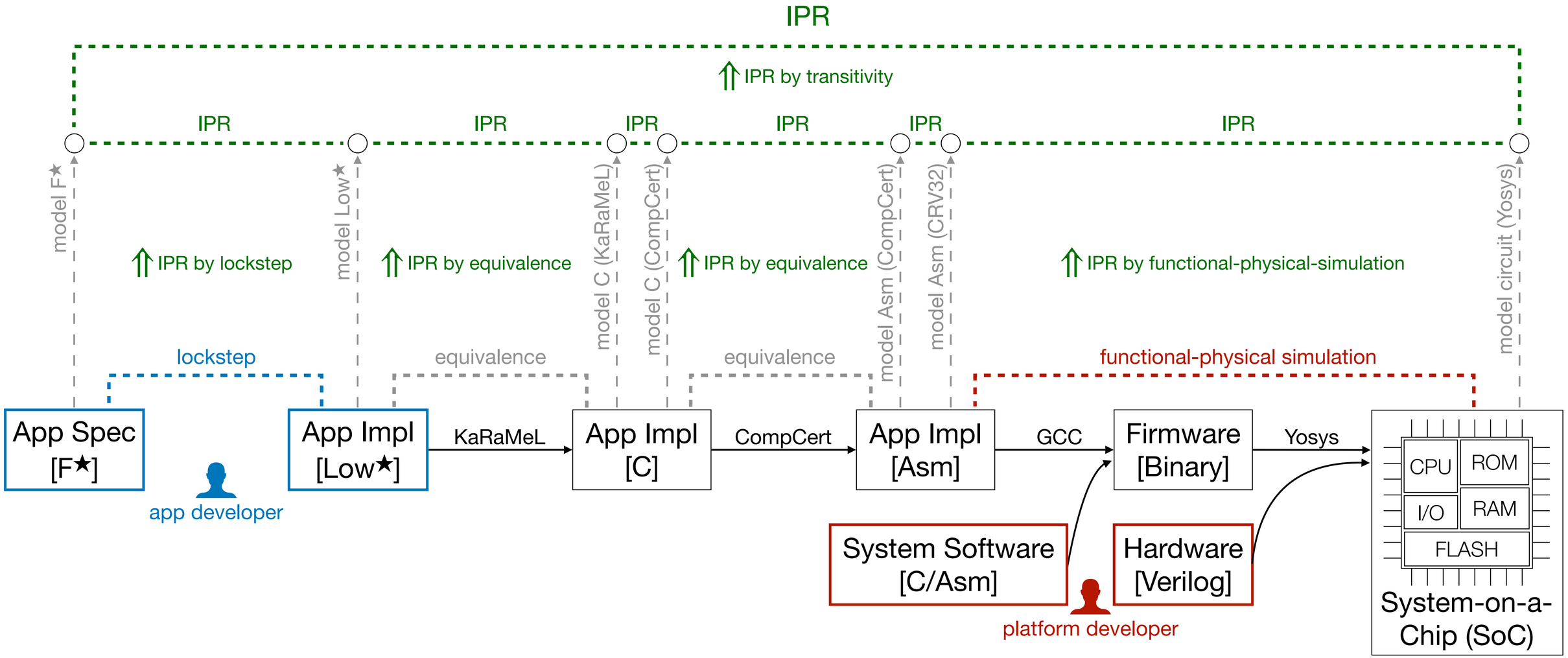
Benefits of the Parfait approach



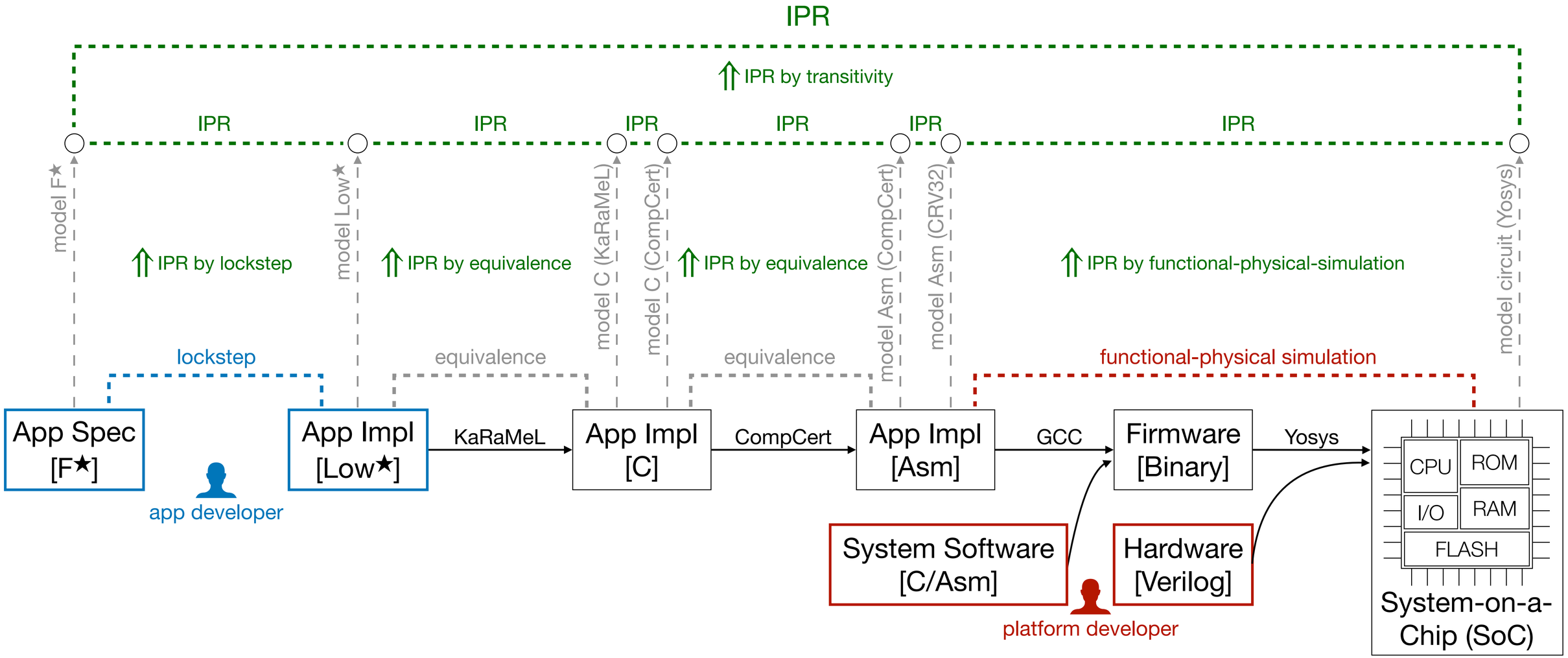
Benefits of the Parfait approach



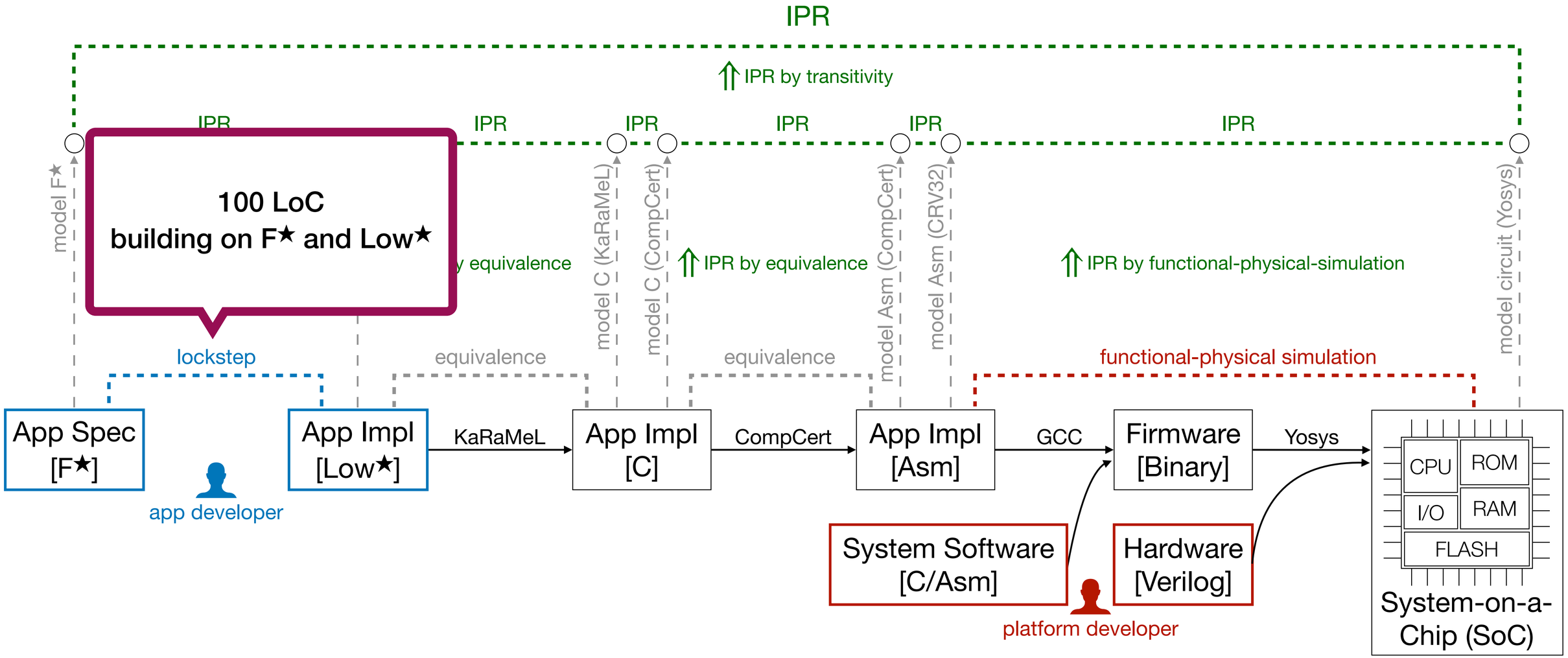
Benefits of the Parfait approach



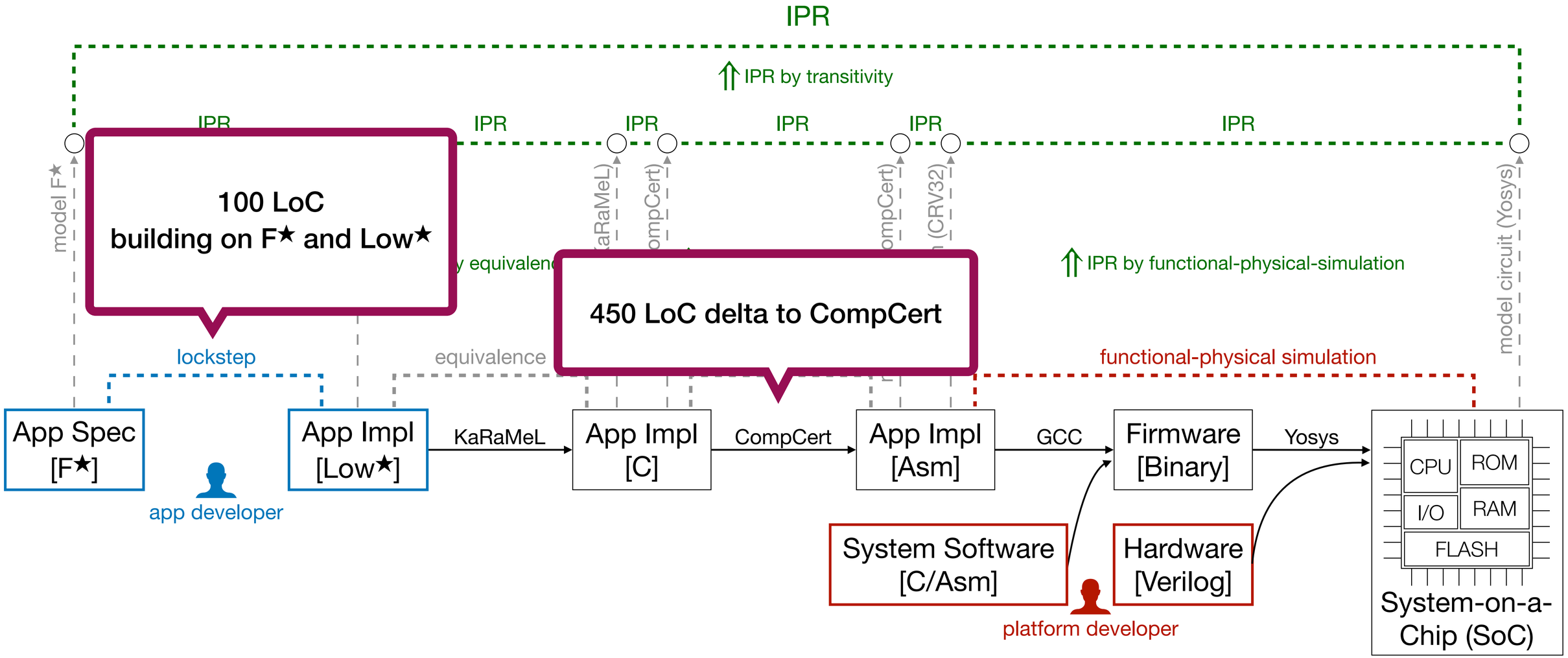
Implementation



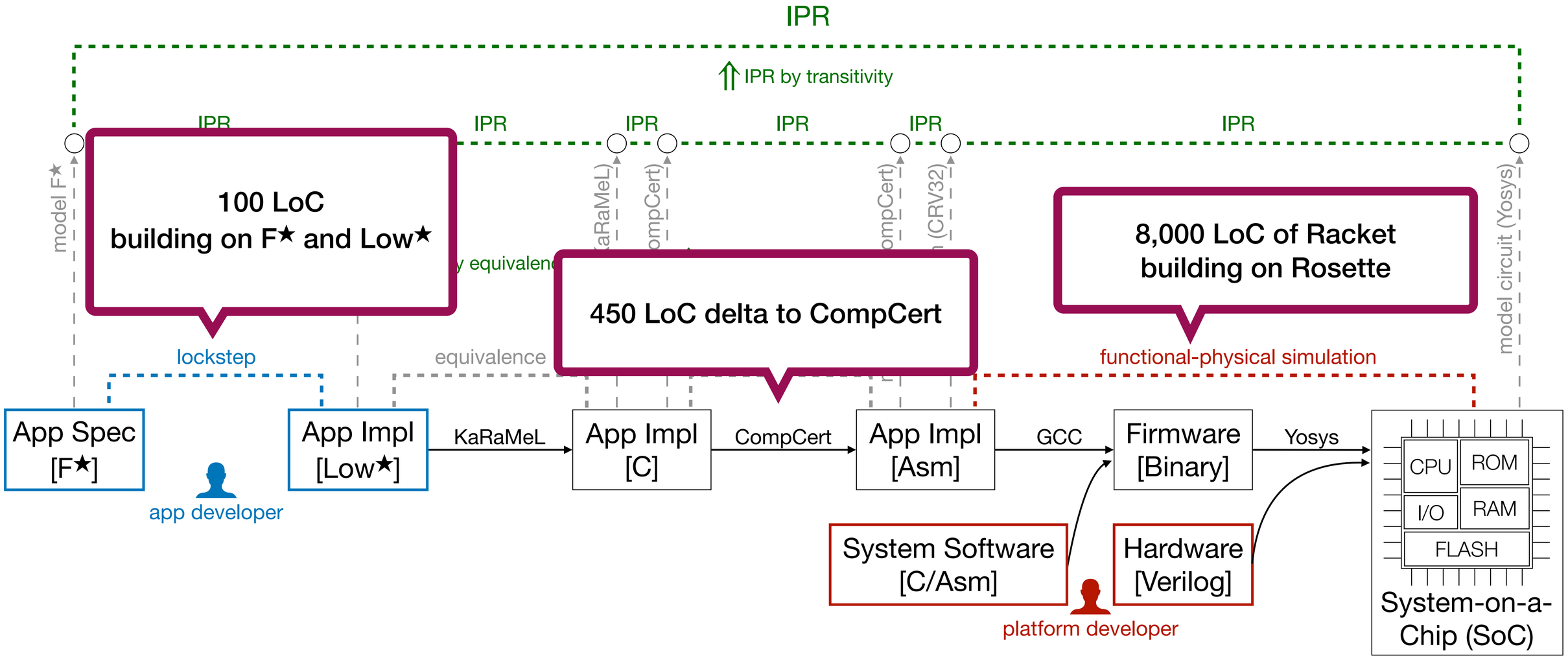
Implementation



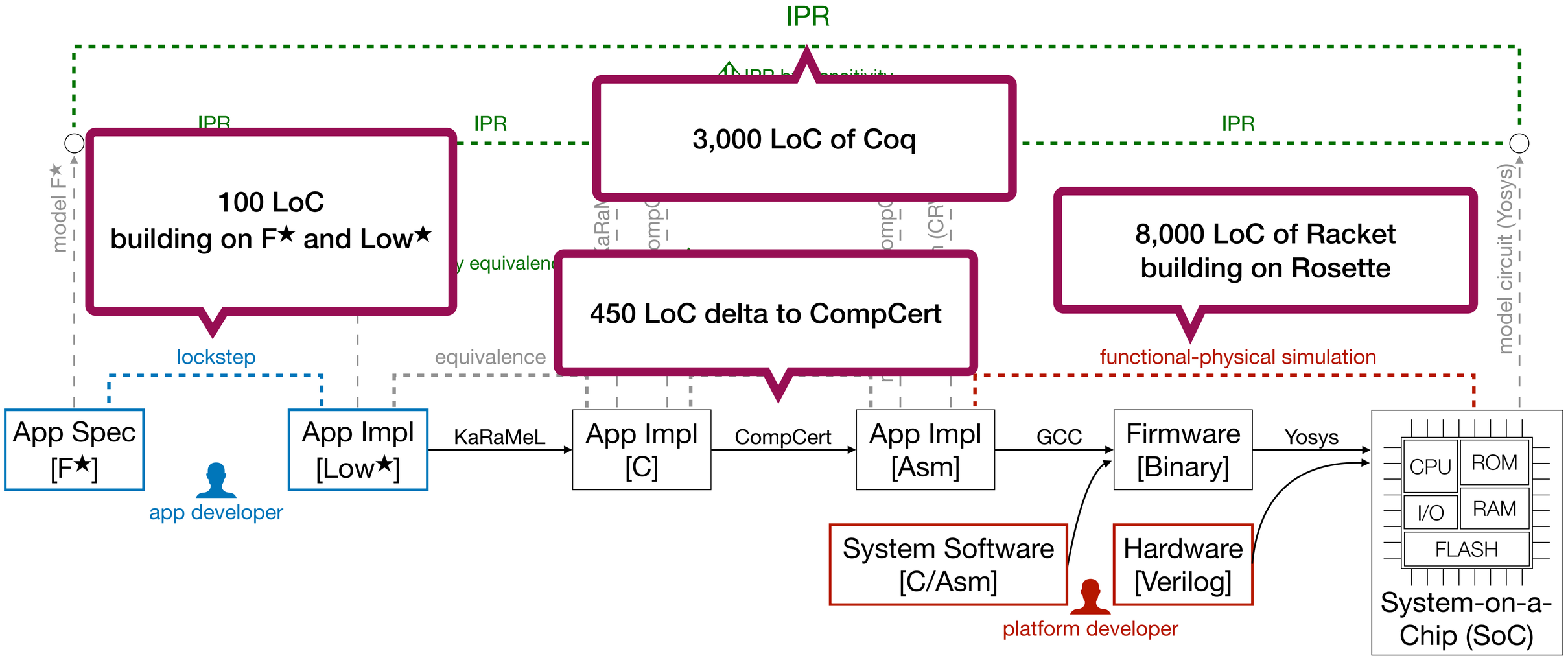
Implementation



Implementation



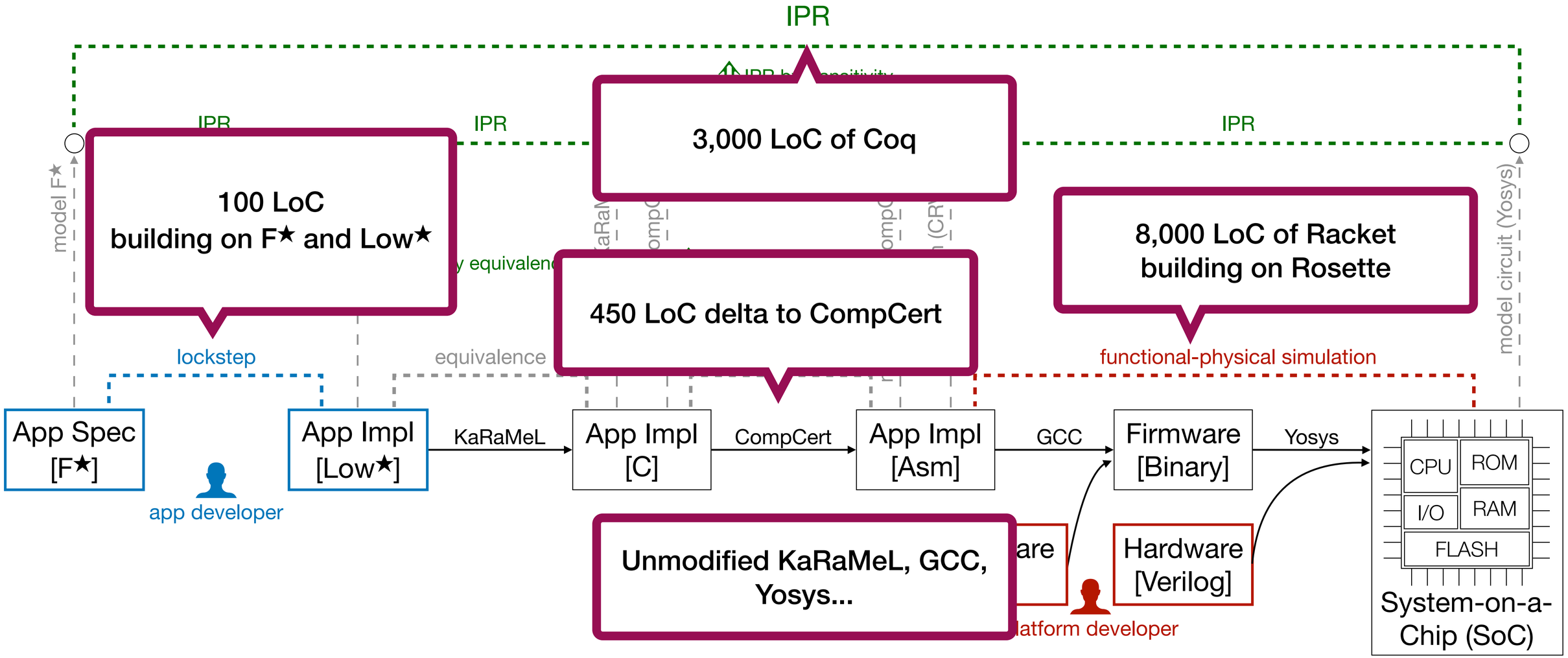
Implementation



Software verification

Hardware verification

Implementation



Parfait case studies

HSM	Spec	Driver	Platform	Implementation	
				Software	Hardware
ECDSA signer	40 LoC	100 LoC	Ibex	2,300 LoC	13,500 LoC
			PicoRV32	2,300 LoC	3,000 LoC
Password hasher	30 LoC	100 LoC	Ibex	1,000 LoC	13,500 LoC
			PicoRV32	1,000 LoC	3,000 LoC

Parfait case studies

HSM	Spec	Driver	Platform	Implementation	
				Software	Hardware
ECDSA signer	40 LoC	100 LoC	Ibex	2,300 LoC	13,500 LoC
			PicoRV32	2,300 LoC	3,000 LoC
Password hasher	30 LoC	100 LoC	Ibex	1,000 LoC	13,500 LoC
			PicoRV32	1,000 LoC	3,000 LoC

Parfait case studies

HSM	Spec	Driver	Platform	Implementation	
				Software	Hardware
ECDSA signer	40 LoC	100 LoC	Ibex	2,300 LoC	13,500 LoC
			PicoRV32	2,300 LoC	3,000 LoC
Password hasher	30 LoC	100 LoC	Ibex	1,000 LoC	13,500 LoC
			PicoRV32	1,000 LoC	3,000 LoC

Parfait case studies

HSM	Spec	Driver	Platform	Implementation	
				Software	Hardware
ECDSA signer	40 LoC	100 LoC	Ibex	2,300 LoC	13,500 LoC
			PicoRV32	2,300 LoC	3,000 LoC
Password hasher	30 LoC	100 LoC	Ibex	1,000 LoC	13,500 LoC
			PicoRV32	1,000 LoC	3,000 LoC

*Spec LOC doesn't include specs from HACL★, which we use without modification

Parfait case studies

HSM	Spec	Driver	Platform	Implementation	
				Software	Hardware
ECDSA signer	40 LoC	100 LoC	Ibex	2,300 LoC	13,500 LoC
			PicoRV32	2,300 LoC	3,000 LoC
Password hasher	30 LoC	100 LoC	Ibex	1,000 LoC	13,500 LoC
			PicoRV32	1,000 LoC	3,000 LoC

Run-time performance

HSM	ECDSA sig/s
Parfait ECDSA/Ibex	1.1
Nitrokey HSM 2	12.5
YubiHSM 2	13.7

Run-time performance

Limited by CompCert

HSM	ECDSA sig/s
Parfait ECDSA/Ibex	1.1
Nitrokey HSM 2	12.5
YubiHSM 2	13.7

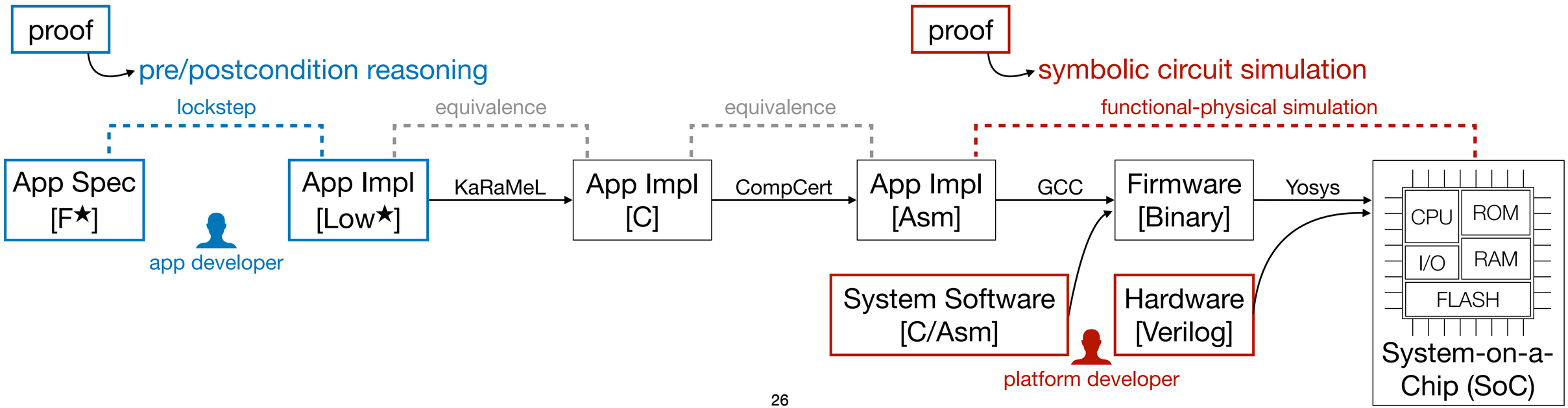
Run-time performance

Limited by CompCert

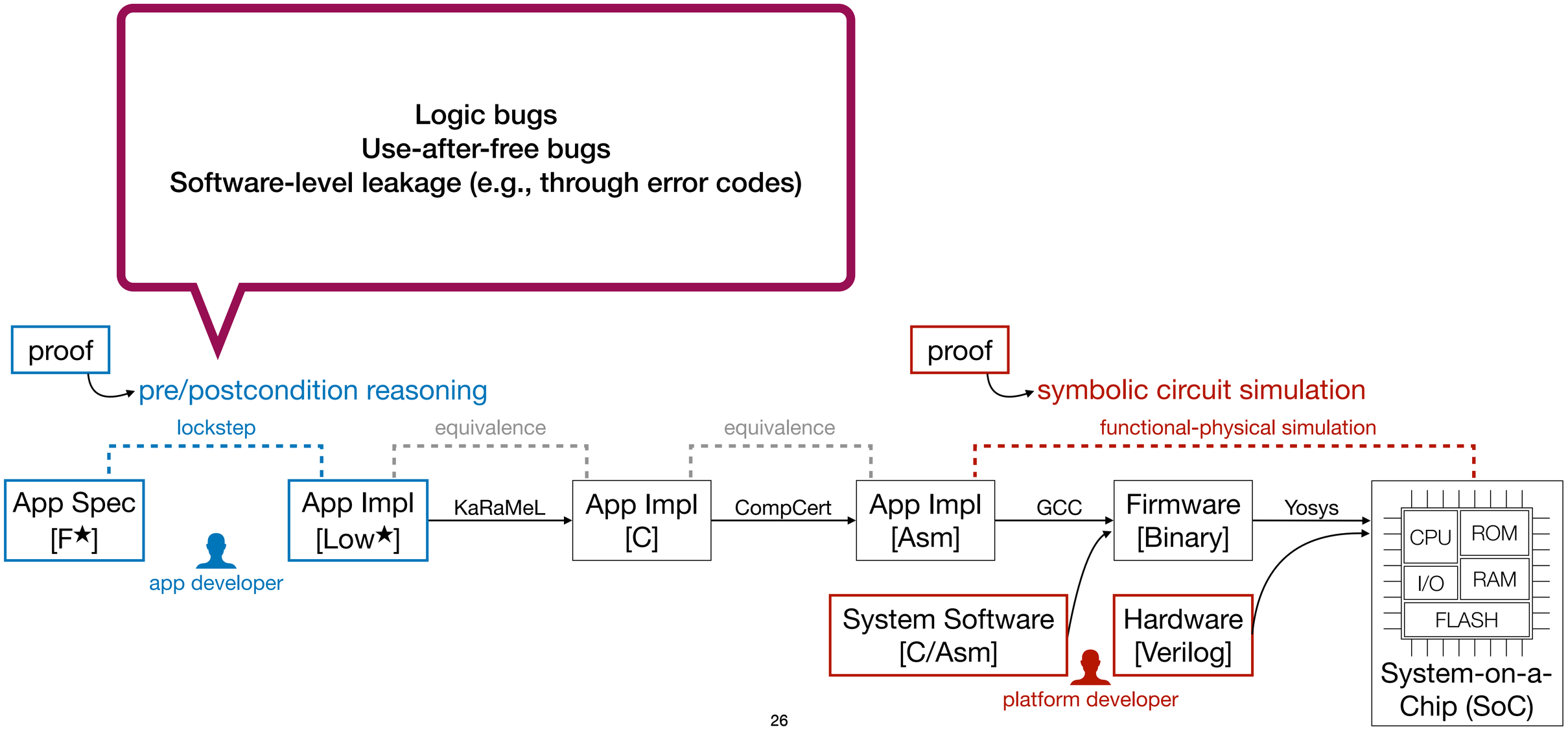
HSM	ECDSA sig/s
Parfait ECDSA/Ibex	1.1
Nitrokey HSM 2	12.5
YubiHSM 2	13.7

If CompCert was as good as GCC: 8.1 sig/s

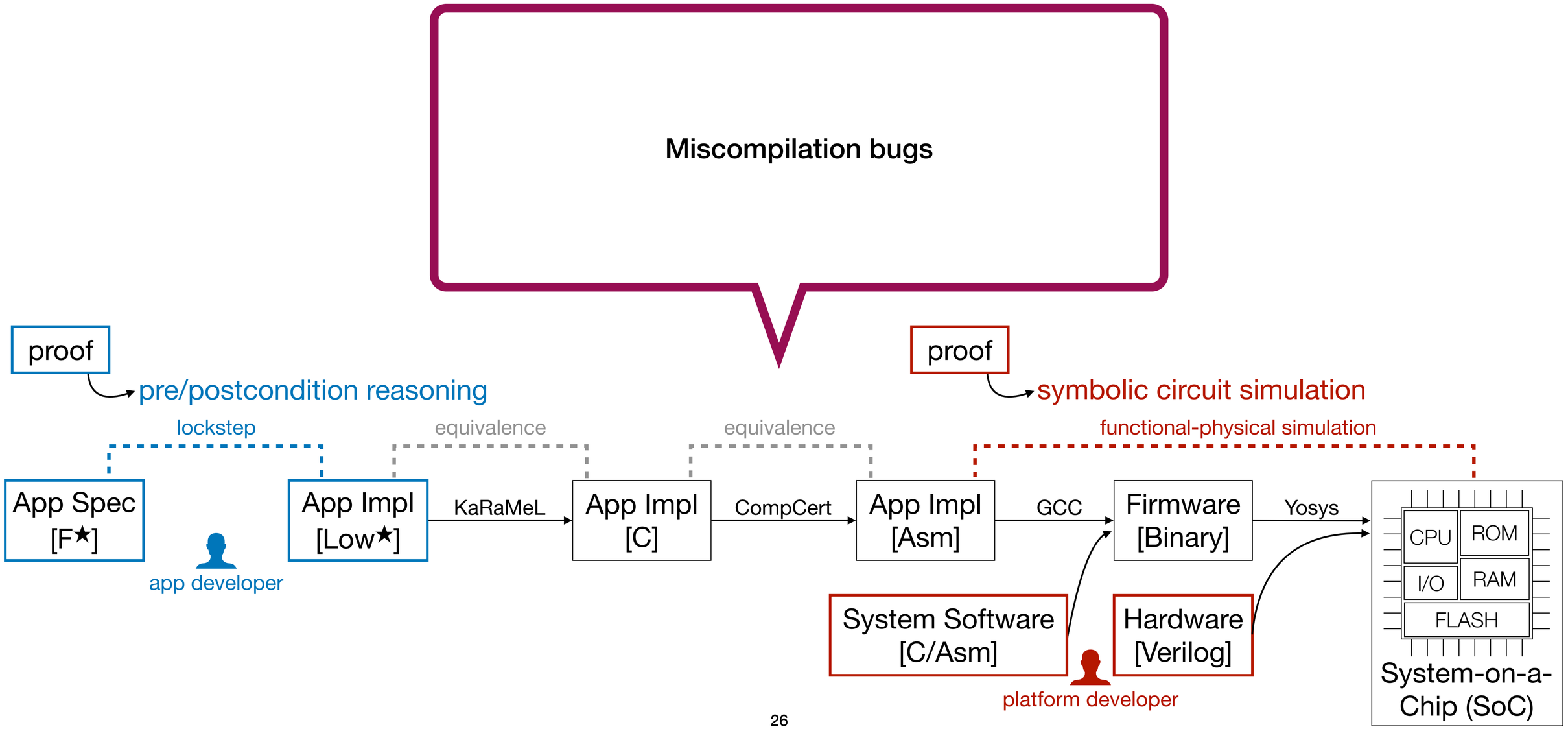
Bugs captured by IPR and Parfait



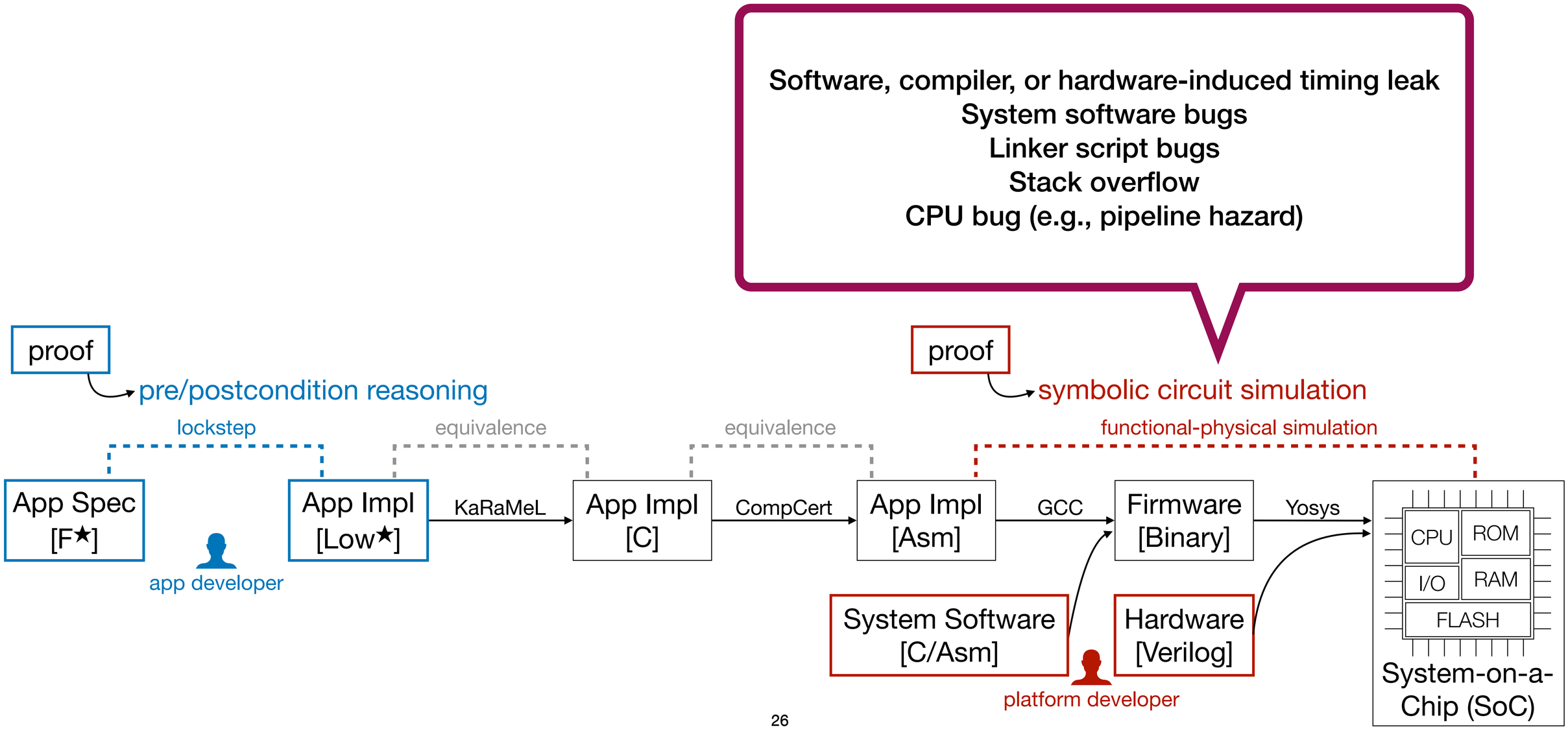
Bugs captured by IPR and Parfait



Bugs captured by IPR and Parfait



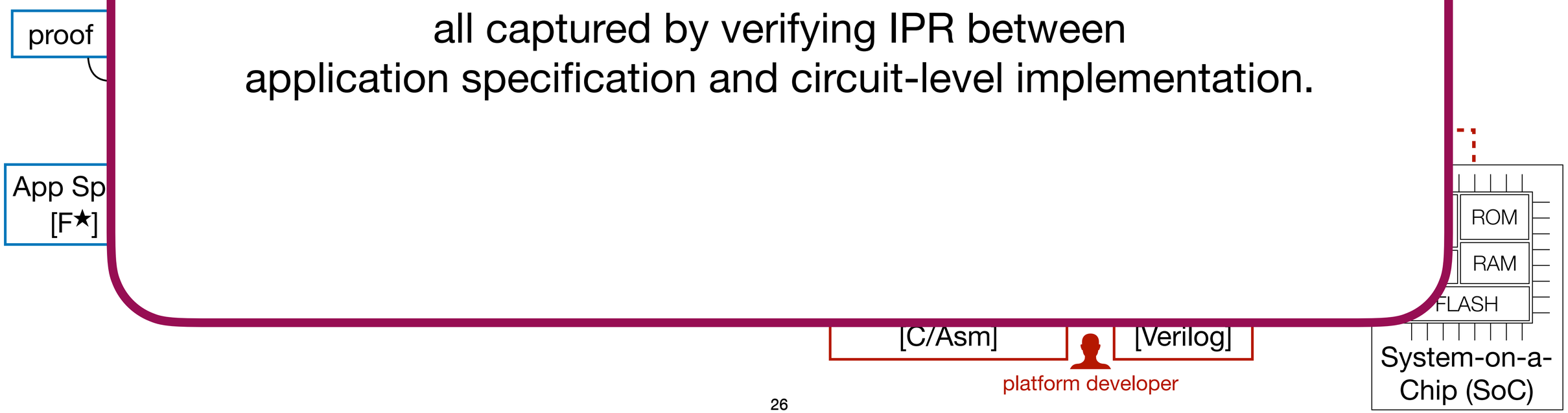
Bugs captured by IPR and Parfait



Bugs captured by IPR and Parfait

No special-case handling for any of these bugs:

all captured by verifying IPR between
application specification and circuit-level implementation.



Evaluation: low effort software proofs

Can implement and verify a new app in a couple hours, reusing existing proofs from HACL★

App	Proof	Dev time
ECDSA signer	500 LoC	-
Password hasher	200 LoC	Δ 2 hours

Evaluation: low effort hardware proofs decoupled from software

Can port to a different CPU in a couple hours of dev time

Make the computer do the hard work

No app-specific proof code

But verification is end-to-end for specific app

	Proof size (LoC)				Verification			
					ECDSA signer		Password hasher	
Platform	Emulator	Hints	Mapping	**Dev time**	Time	Cycles/s	Time	Cycles/s
Ibex	50	250	10	-	80 hrs	304	0.10 hrs	289
PicoRV32			10	Δ 2 hours	100 hrs	671	0.14 hrs	588

Evaluation: low effort hardware proofs decoupled from software

Can port to a different CPU in a couple hours of dev time

Make the computer do the hard work

No app-specific proof code

But verification is end-to-end for specific app

	Proof size (LoC)				Verification			
					ECDSA signer		Password hasher	
Platform	Emulator	Hints	Mapping	**Dev time**	Time	Cycles/s	Time	Cycles/s
Ibex	50	250	10	-	80 hrs	304	0.10 hrs	289
PicoRV32			10	Δ 2 hours	100 hrs	671	0.14 hrs	588

Evaluation: low effort hardware proofs decoupled from software

Can port to a different CPU in a couple hours of dev time

Make the computer do the hard work

No app-specific proof code

But verification is end-to-end for specific app

	Proof size (LoC)				Verification			
					ECDSA signer		Password hasher	
Platform	Emulator	Hints	Mapping	**Dev time**	Time	Cycles/s	Time	Cycles/s
Ibex	50	250	10	-	80 hrs	304	0.10 hrs	289
PicoRV32			10	Δ 2 hours	100 hrs	671	0.14 hrs	588

Evaluation: low effort hardware proofs decoupled from software

Can port to a different CPU in a couple hours of dev time

Make the computer do the hard work

No app-specific proof code

But verification is end-to-end for specific app

	Proof size (LoC)				Verification			
					ECDSA signer		Password hasher	
Platform	Emulator	Hints	Mapping	**Dev time**	Time	Cycles/s	Time	Cycles/s
Ibex	50	250	10	-	80 hrs	304	0.10 hrs	289
PicoRV32			10	Δ 2 hours	100 hrs	671	0.14 hrs	588

Related work

Hardware/software co-verification: Bedrock2 [PLDI'21, PLDI'24], CakeML [PLDI'19], Knox [OSDI'22]

> Focused on correctness, not confidentiality (including side-channel leakage)
> Knox does not scale to HSMs with software like public-key crypto

Leakage models: HACL★ [CCS'17], ct-verif [Sec'16], SideTrail [VSTTE'18], CompCert-CT [POPL'19], LeaVe [CCS'23], ...

> No end-to-end (application-level spec to RTL) results

Parfait is the first to verify non-leakage from app spec to RTL with modular proofs

Conclusion

Information-Preserving Refinement (IPR) formalizes correctness, security, and non-leakage

Transitive IPR: scaling proofs of IPR with modularity

Proof techniques for IPR: proving IPR across the software/hardware stack

Parfait verification framework: implements these techniques

Application of Parfait to HSMs

anish.io/parfait