

Colloquium in honor of Martin Abadi



Geppetto: **Versatile Verifiable Computation**

Cédric Fournet

with Craig Costello, Antoine Delignat-Lavaud*, Jon Howell, Chantal Keller,
Markulf Kohlweiss, Benjamin Kreuter*, Vincent Laporte*,
Michael Naehrig, Bryan Parno, Samee Zahur*

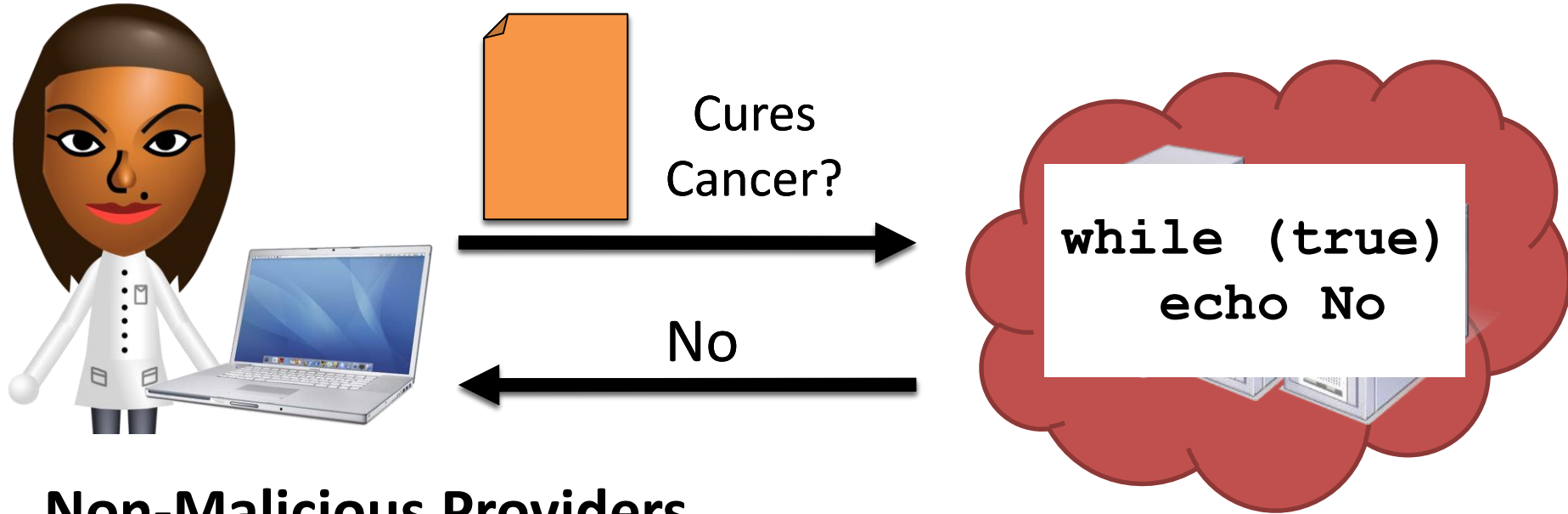


Microsoft®

Research

<http://eprint.iacr.org/2014/976>

Outsourcing Computation



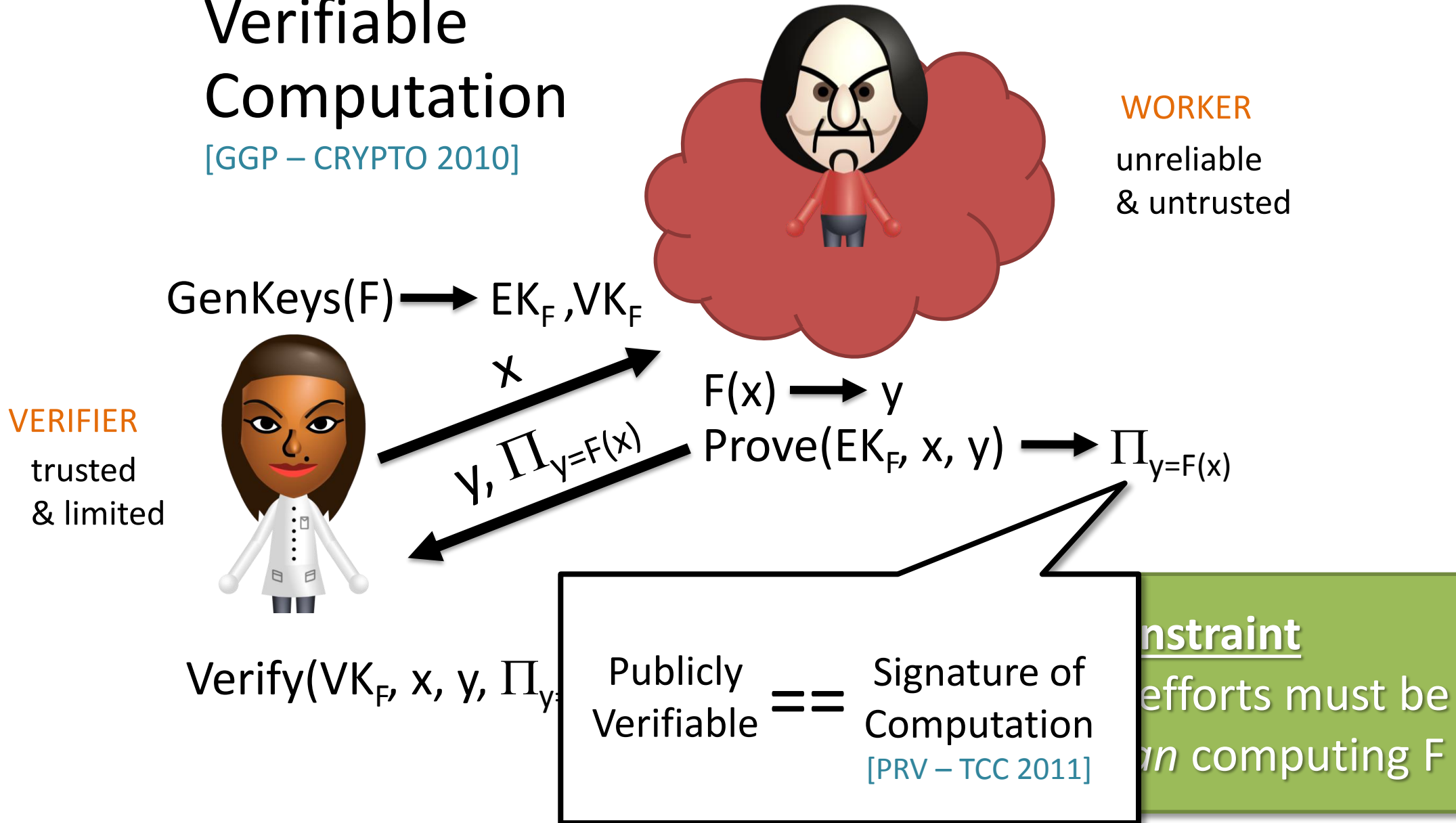
Non-Malicious Providers



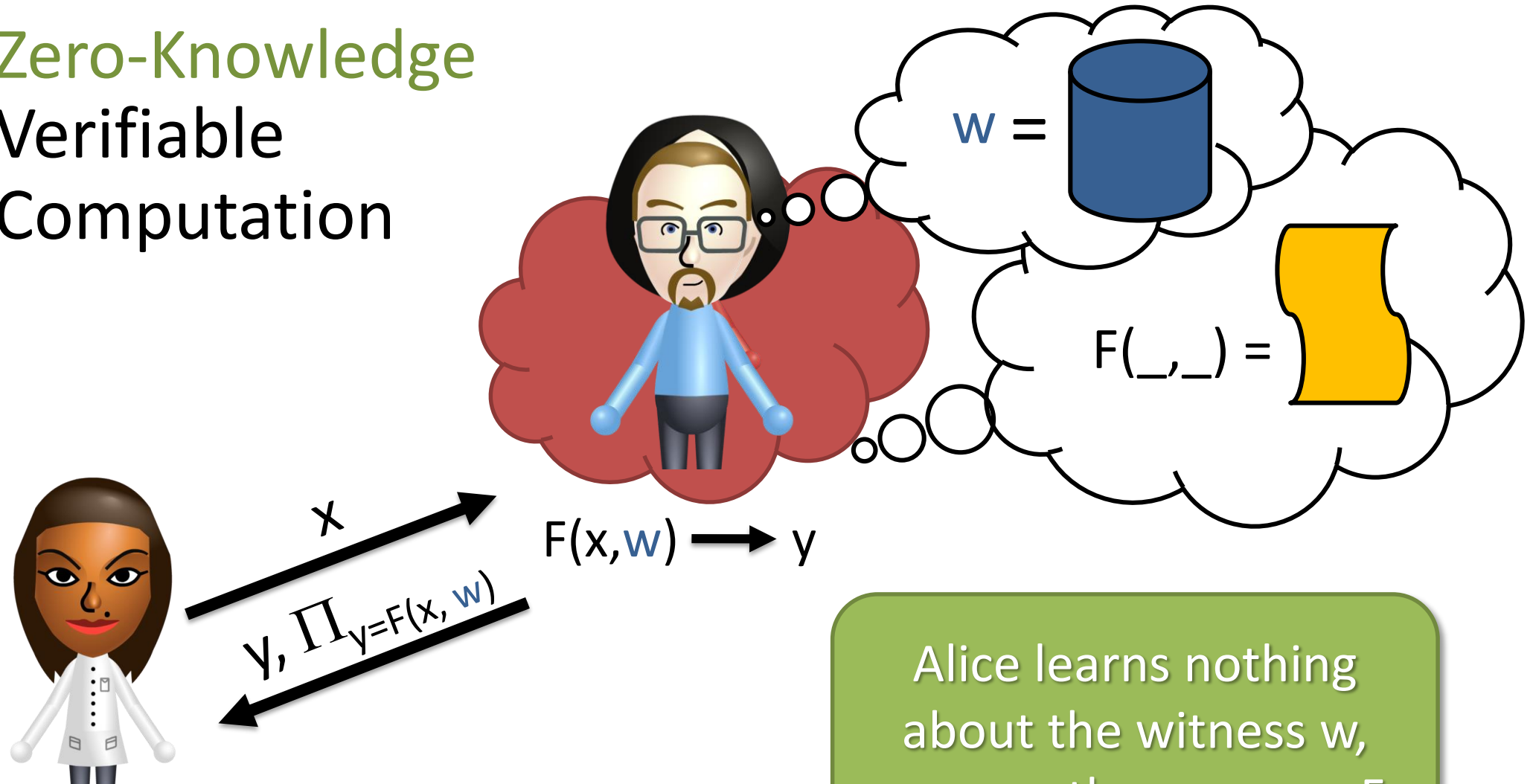
- Verifiable results:
 - Encourage cloud adoption
 - Command a premium
 - Help shed liability

Verifiable Computation

[GGP – CRYPTO 2010]



Zero-Knowledge Verifiable Computation



Alice learns nothing
about the witness w ,
or even the program F

Secure circuit evaluation

A protocol based on hiding information from an oracle*

Martín Abadi

DEC Systems Research Center

130 Lytton Avenue

Palo Alto, CA 94301

Joan Feigenbaum

AT&T Bell Laboratories

600 Mountain Avenue

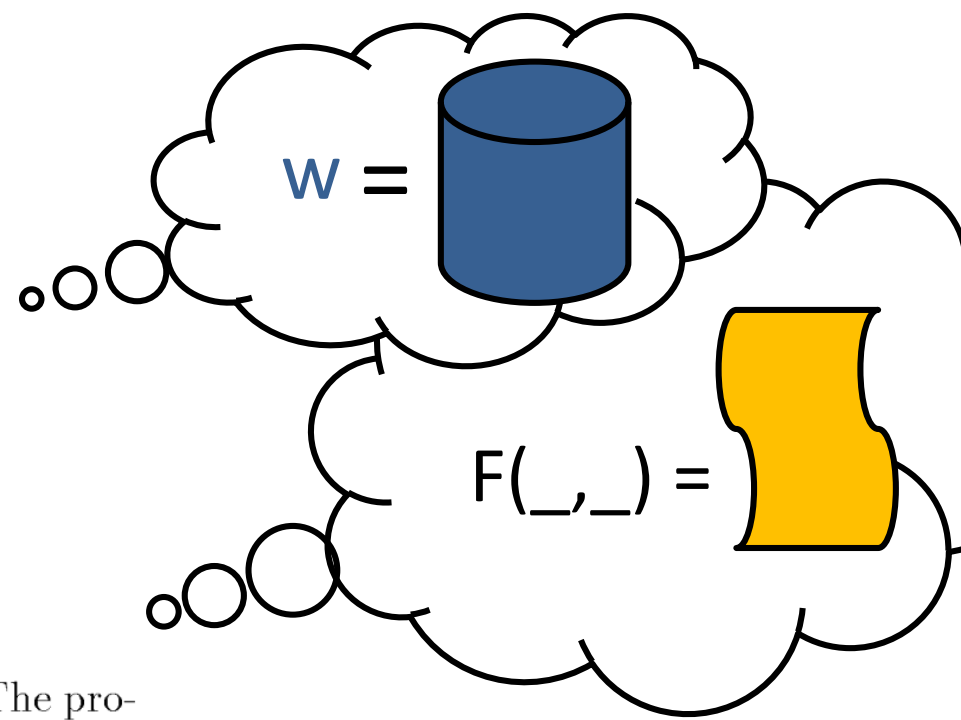
Murray Hill, NJ 07974

Abstract: We present a simple protocol for two-player *secure circuit evaluation*. The protocol enables players C and D to cooperate in the computation of $f(x)$ while D conceals her data x from C and C conceals his circuit for f from D. The protocol is based on the technique of *hiding information from an oracle* [Abadi, Feigenbaum, Kilian, *J. Comput. System Sci.* 39(1):21–50, August, 1989].

1. Introduction

J. Cryptology 1990

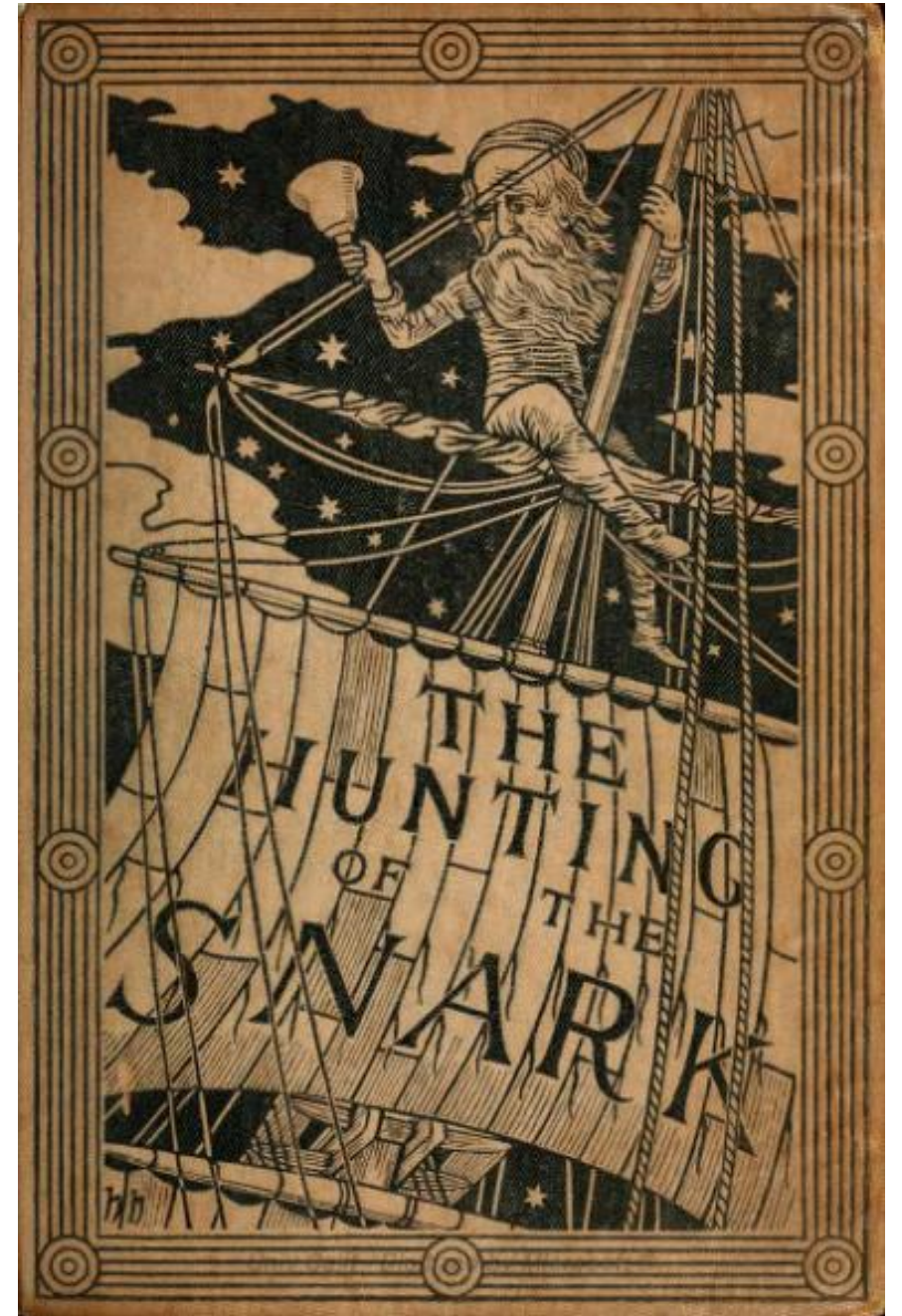
This paper describes a protocol for two-player *secure circuit evaluation*. We think of player C as a computer that runs a secret program for f and player D as another computer with some confidential data x . Suppose that D wishes to compute $f(x)$ but does not have an algorithm for f , and that C is willing to let D use his algorithm but does not want to reveal it. Alternatively, suppose that C has an algorithm, and possibly some confidential



What are SNARKs?

- Some Snarks are highly dangerous Boojums (Lewis Carroll)
- Others are Succinct Non-Interactive Arguments of Knowledge
- Proofs (or arguments) are **constant-size**

Quadratic Span Programs and Succinct NIZKs without PCPs
Rosario Gennaro* Craig Gentry† Bryan Parno‡ Mariana Raykova§



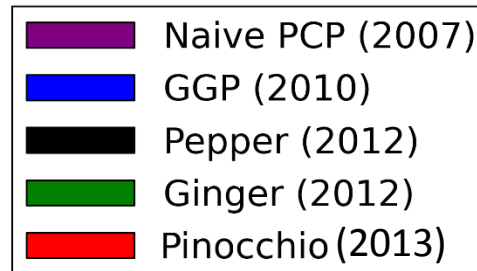
Relative Performance

100x100 matrix mult.



Verifier Latency

Worker Latency



Time (sec)

10²²
10²⁰
10¹²
10¹⁰
10⁸
10⁶
10⁴
10²
10⁰
10⁻²

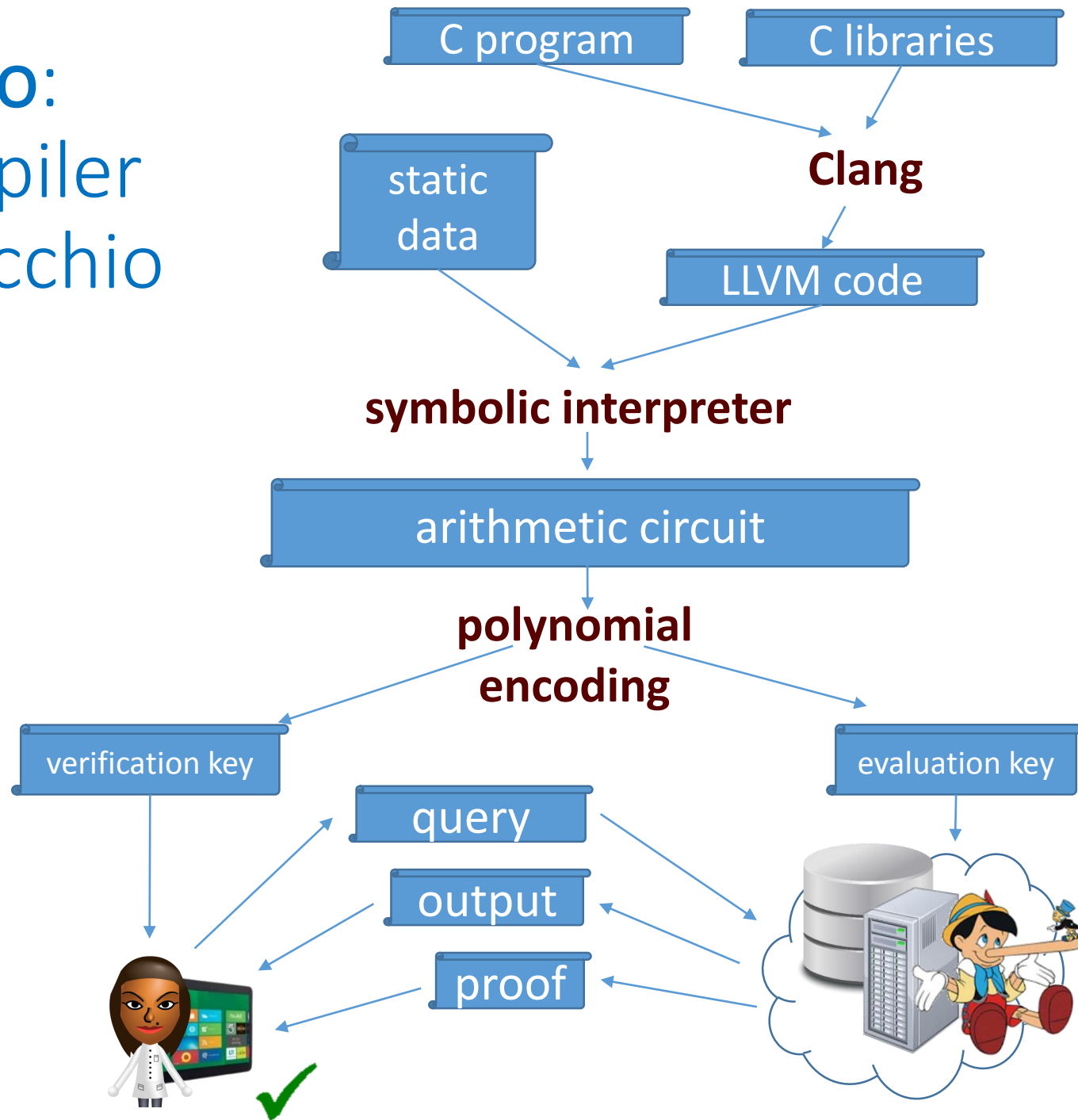
fell 23 orders of magnitude in

15 ms

Building on Pinocchio

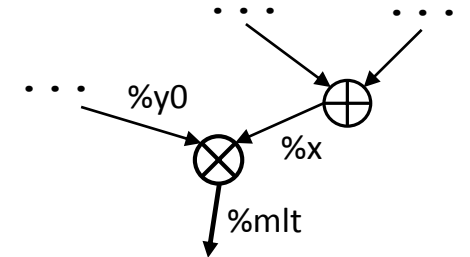
- Optimizing for set operations
[Kosba et al. '14]
- Optimizing for authenticated data
[Backes, Fiore, Reischuk '15]
- More expressivity
[Ben-Sasson et al. '14] [Wahby et al. '15]
- Key generation via SMC
[Ben-Sasson et al. '15]
- Applications
 - BitCoin
[Danezis et al. '13] [Ben-Sasson et al. '14]
 - MapReduce
[Braun et al. '14] [Ben-Sasson et al. '15]

Geppetto: a C compiler for Pinocchio



```
void main() {  
    ...  
    x = b[i] + d[j];  
    y *= x;  
    return y; }
```

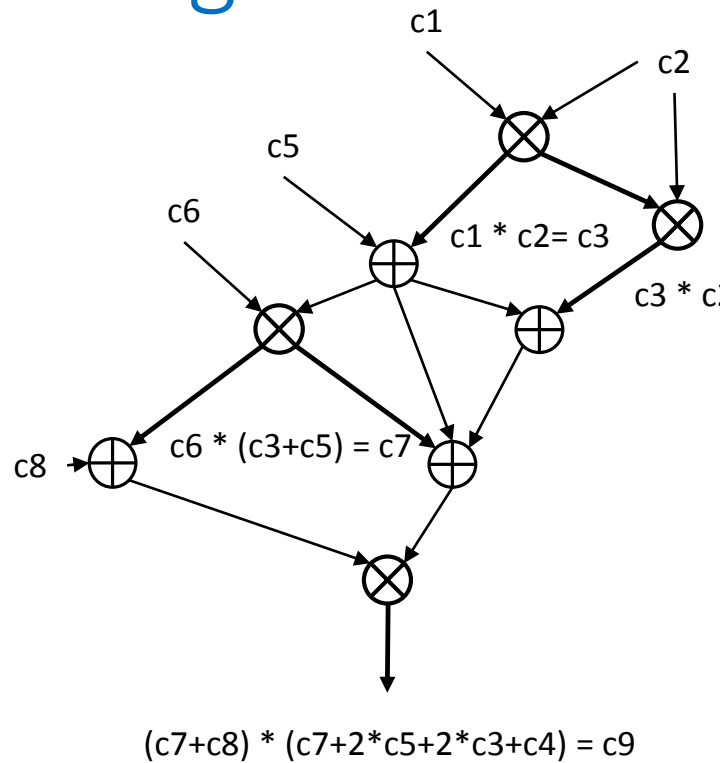
```
define i32 @main() #1 {  
entry:  
    ...  
    %y0 = load i32* %y.addr  
    %mlt = mul nsw i32 %x, %y0  
    ret i32 %mlt }
```



```
bc96 a736 6e97 8bfc 957f 5857 d4a6 2770  
f3cb f2e0 d5f5 de09 f46a 7d00 e8db 0e0a  
7430 a958 9875 9d7a 5d5e 0cec 955d 13c5  
b605 be6d 1685 72fa c3e9 dc09 c7dc cc05  
ca47 04cc e739 663f 3c4a 9dea c74b 7c5b
```

Quadratic Arithmetic Programming (QAP)

Compilation involves coding every step of the C program into **quadratic equations** in a large prime field.



```
void outsource(struct Input *input,
               struct Output *output)
{
    int c1 = input->c1; int c2 = input->c2;
    int c5 = input->c5; int c6 = input->c6;
    int c8 = input->c8;

    output->c9 =
        (c6 * (c1*c2 + c5) + c8) *
        (c6 * (c1*c2 + c5) + 2*c5 + 2*c1*c2
         + c1*c2*c2);
}
```

Quadratic Arithmetic Programming (QAP)

Compilation involves coding every step of the C program into **quadratic equations** in a large prime field.

$$\bigwedge_{r=1..d} (\vec{v}_r \cdot \vec{c})(\vec{w}_r \cdot \vec{c}) = \vec{y}_r \cdot \vec{c}$$

$$\Leftrightarrow \bigwedge_{x=1..d} (\vec{v}[x] \cdot \vec{c})(\vec{w}[x] \cdot \vec{c}) = \vec{y}[x] \cdot \vec{c}$$

$$\Leftrightarrow d(x) * \prod_{r=1..d} (x - r) = (\vec{v}[x] \cdot \vec{c})(\vec{w}[x] \cdot \vec{c}) - \vec{y}[x] \cdot \vec{c}$$

Idea: prove this *single* polynomial equation at a *single* random point $x = s$, then apply Schwartz-Zippel Lemma

What about other program operations?

Compile to quadratic arithmetic!

Source
(C code)

```
{ b = (x == y);  
  ...
```

```
  r = x && 0xFFFF;  
  ...
```

```
}
```

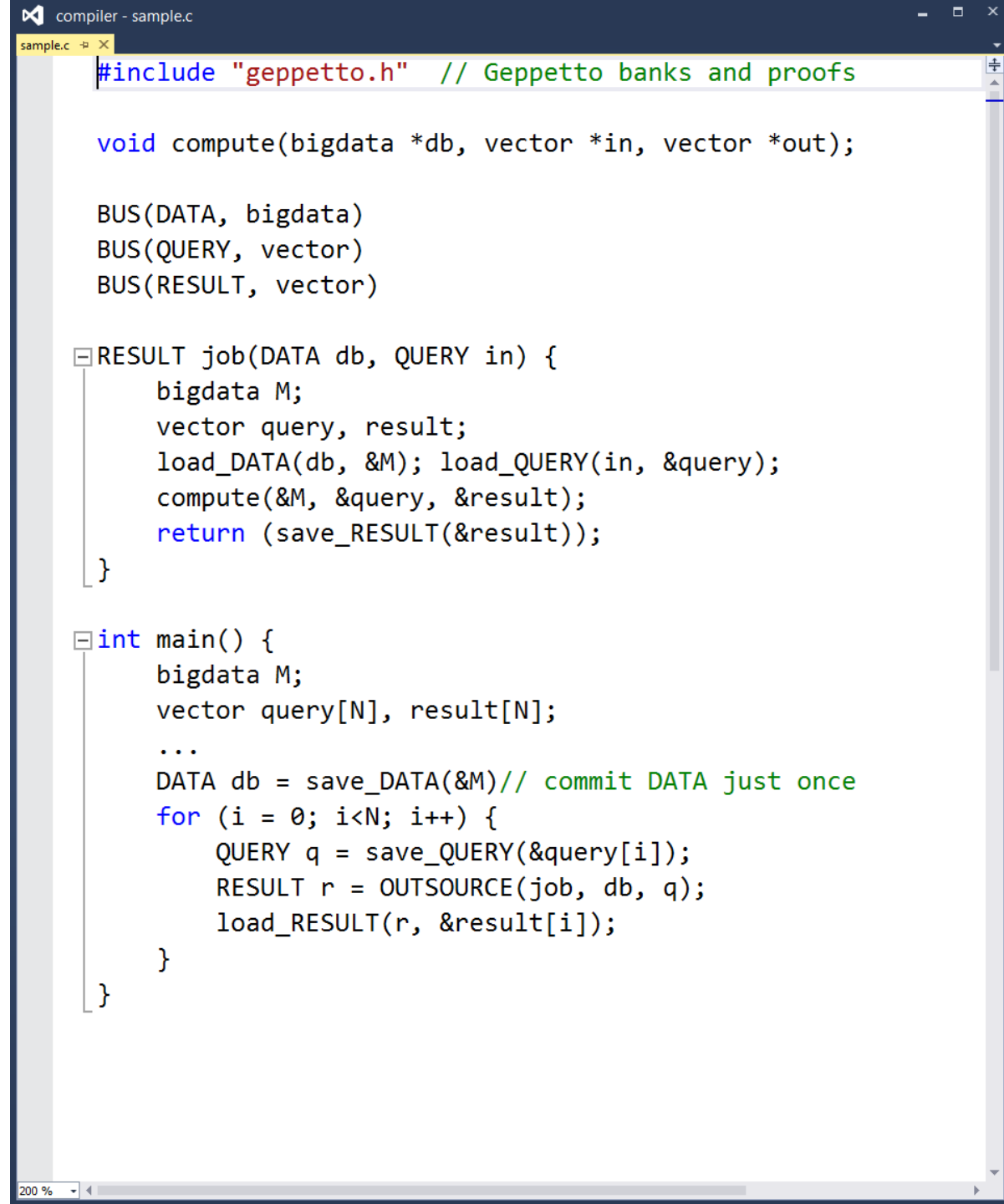

$$\begin{aligned} & (y - x) * z = 1 - b \\ \wedge & (y - x) * b = 0 \end{aligned}$$


$$\begin{aligned} & \bigwedge_{i=0}^{31} (b_i - 1) * b_i = 0 \\ \wedge & x = \sum_{i=0}^{31} 2^i b_i \\ \wedge & r = \sum_{i=0}^{15} 2^i b_i \end{aligned}$$

Programming model: low-level verifiable C

We reuse C experience & tools

- programmers explicitly manage outsourced calls (without any crypto)
- **outsourced calls $\approx$ RPCs**



```
compiler - sample.c
sample.c
#include "geppetto.h" // Geppetto banks and proofs

void compute(bigdata *db, vector *in, vector *out);

BUS(DATA, bigdata)
BUS(QUERY, vector)
BUS(RESULT, vector)

RESULT job(DATA db, QUERY in) {
    bigdata M;
    vector query, result;
    load_DATA(db, &M); load_QUERY(in, &query);
    compute(&M, &query, &result);
    return (save_RESULT(&result));
}

int main() {
    bigdata M;
    vector query[N], result[N];
    ...
    DATA db = save_DATA(&M) // commit DATA just once
    for (i = 0; i < N; i++) {
        QUERY q = save_QUERY(&query[i]);
        RESULT r = OUTSOURCE(job, db, q);
        load_RESULT(r, &result[i]);
    }
}
```

```
$ clang sample.c -o sample.exe
```

```
$ sample.exe
```

```
// runs reference code & generates sample.trace
```

```
The answer is 42.
```

```
$ clang -DQAP -S sample.c -emit-llvm -o sample.s
```

```
$ geppetto -load sample.s -schedule sample.trace  
-compile -emit verify.c -keygen
```

```
// generates keys & native verifier
```

```
Compiled job to 13168 equations on 11287 vars.
```

```
$ geppetto -load sample.s -prove
```

```
// runs whole program
```

```
// generating commitments and proofs
```

```
The answer is 42. (slower)
```

```
$ clang -DVERIFY verify.c sample.c -o verify.exe
```

```
$ verify.exe
```

```
// runs whole program, loading & verifying
```

```
// commitments and proofs instead of
```

```
// running outsourced calls.
```

```
The answer is 42. (faster)
```

```
sample.c -> X
#include "geppetto.h" // Geppetto banks and proofs

void compute(bigdata *db, vector *in, vector *out);

BUS(DATA, bigdata)
BUS(QUERY, vector)
BUS(RESULT, vector)

RESULT job(DATA db, QUERY in) {
    bigdata M;
    vector query, result;
    load_DATA(db, &M); load_QUERY(in, &query);
    compute(&M, &query, &result);
    return (save_RESULT(&result));
}

int main() {
    bigdata M;
    vector query[N], result[N];
    ...
    DATA db = save_DATA(&M) // commit DATA just once
    for (i = 0; i < N; i++) {
        QUERY q = save_QUERY(&query[i]);
        RESULT r = OUTSOURCE(job, db, q);
        load_RESULT(r, &result[i]);
    }
}
```

From LLVM to QAPs: Twin symbolic interpreters

Compile-time:

- We compile outsourced functions, and generate keys.
- We use a precise integer abstraction, keeping track of **range** and **semantics** for each unknown integer
- We emit a stream of **quadratic equations** for their operations

We keep QAPs implicit:
symbolic interpretation is faster
& more general than emitting and
evaluating arithmetic circuits

Prove-time:

- We run the whole program, generating commitments and proofs.
- We use a simpler integer abstraction, still keeping track of known vs 'unknown' integers (although we have both)
- We emit a stream of **witnesses** for their operations

Verification-time:

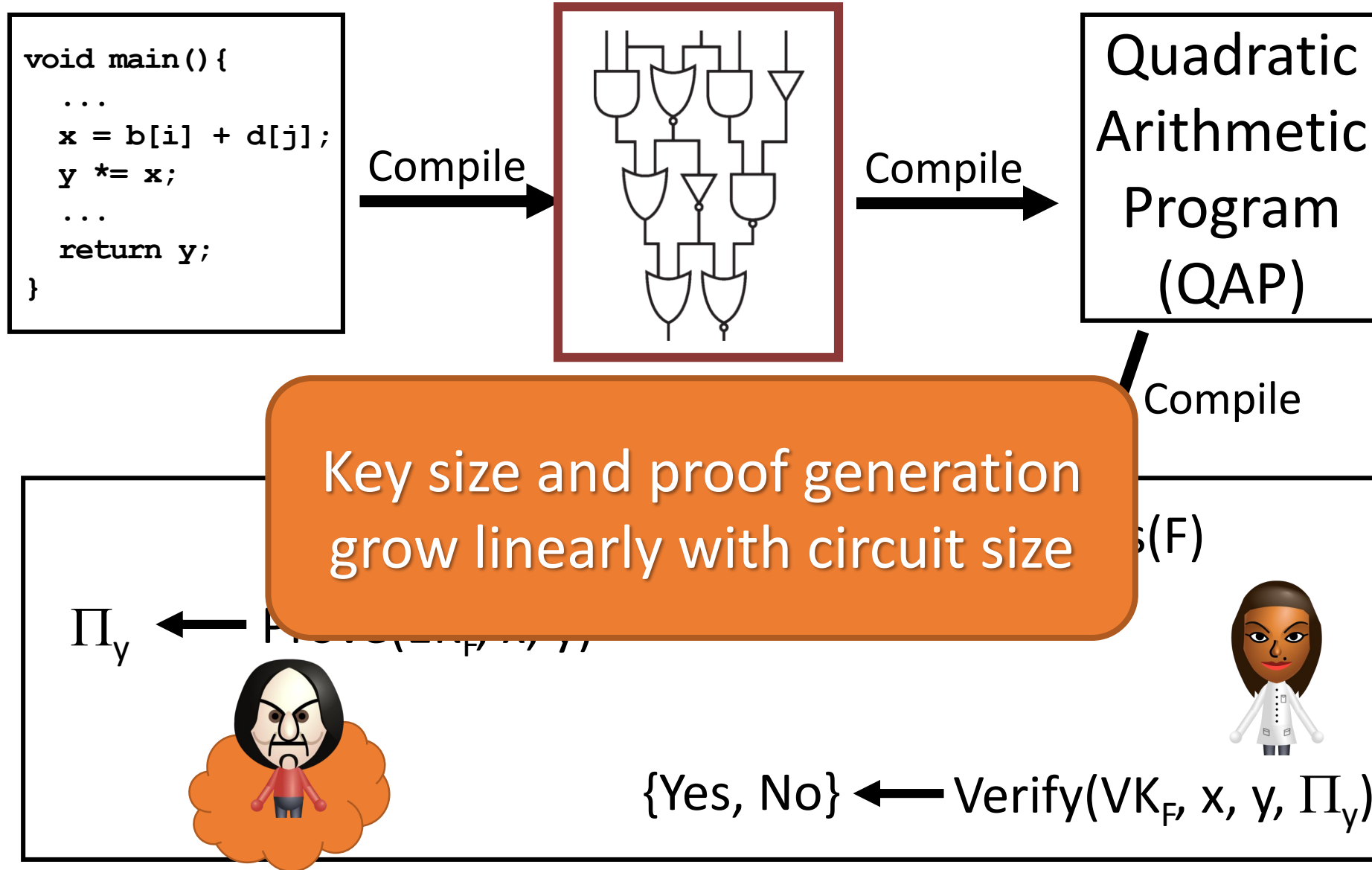
- We run the whole program, verifying evidence instead of running outsourced functions

Geppetto: technical overview

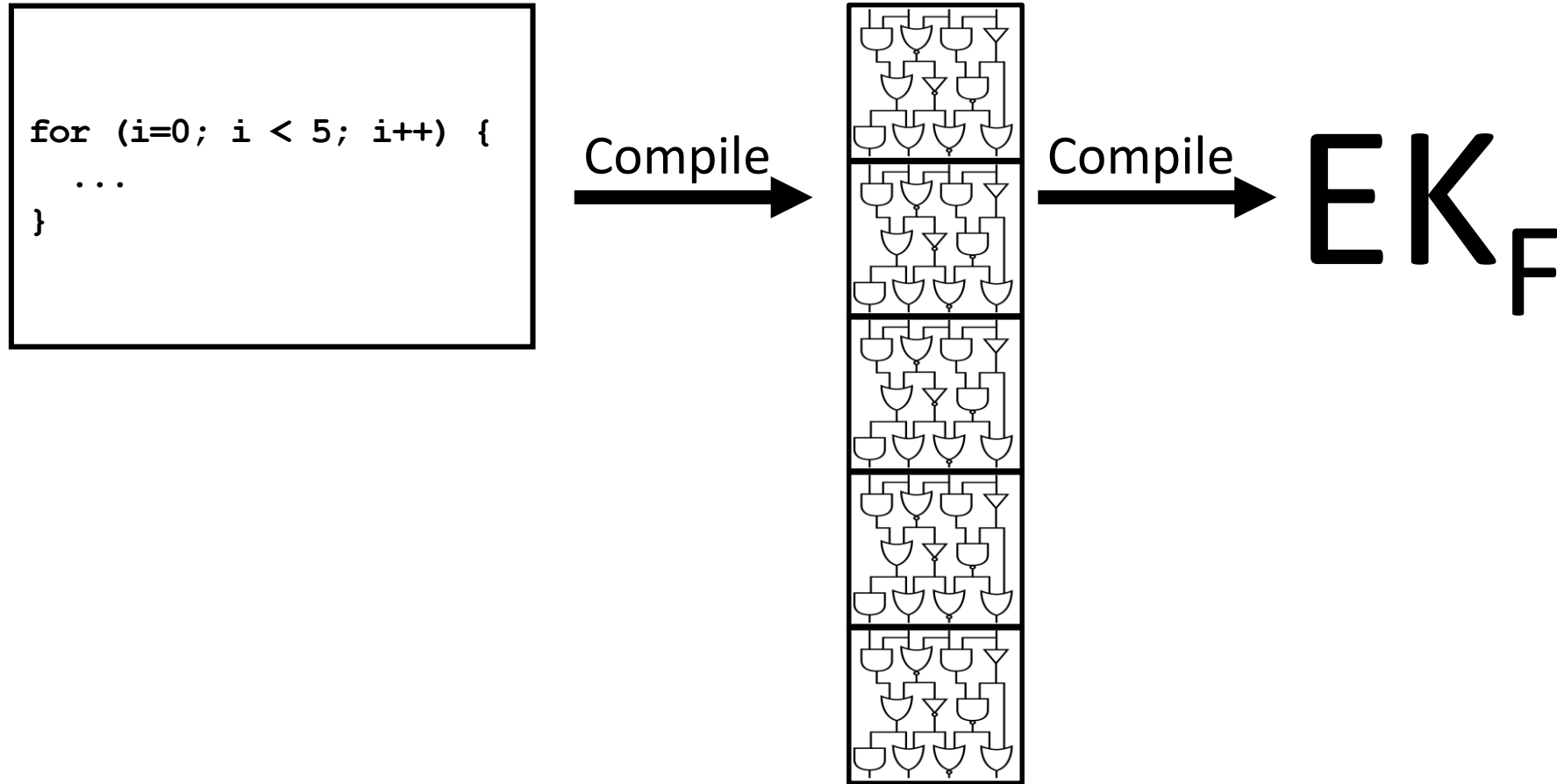
New, interlocked techniques
for flexibility & efficiency

- **Multi-step proofs**
to verify large computations
& amortize costs
- **Bootstrapping**
to compress multi-step proofs
into a single proof of proofs.
- **Crypto Libraries**
for security applications

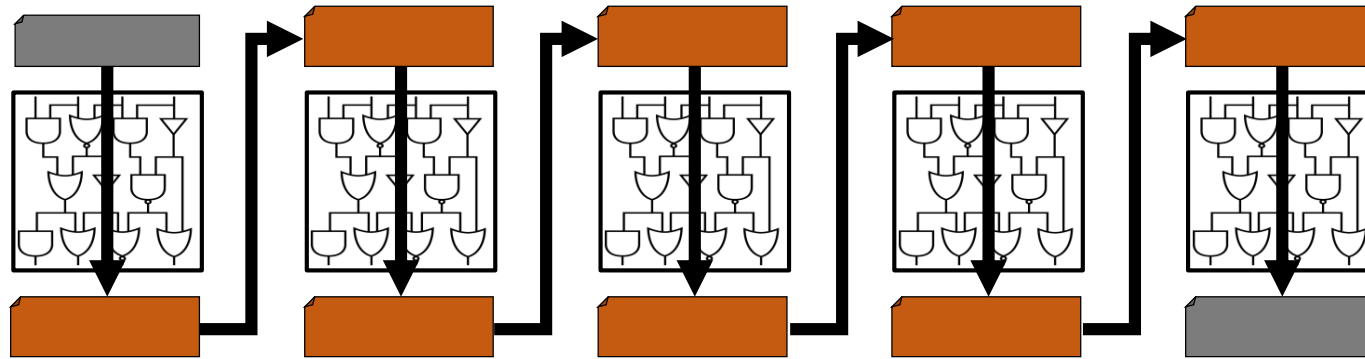
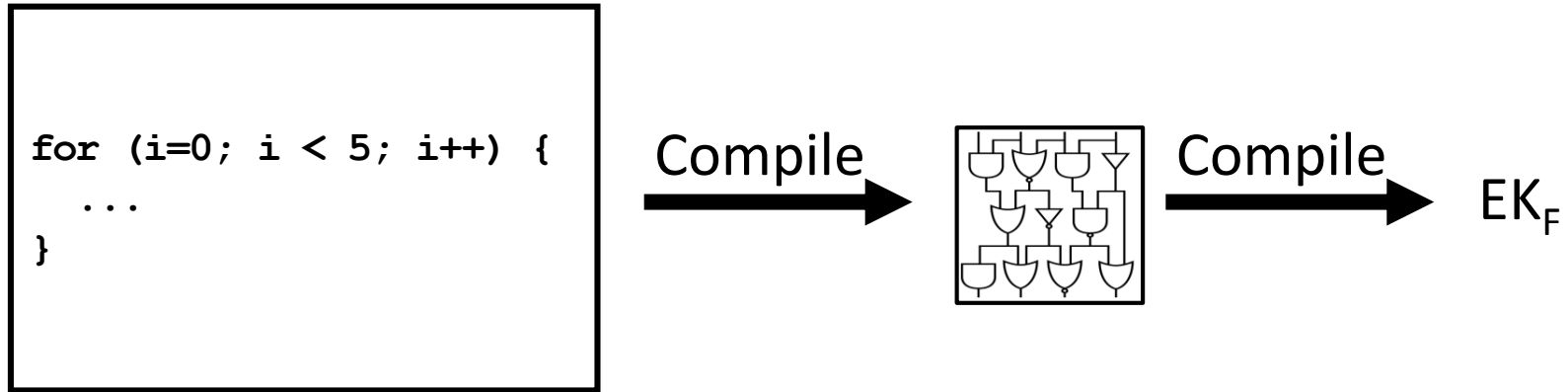
Background: Pinocchio's Pipeline



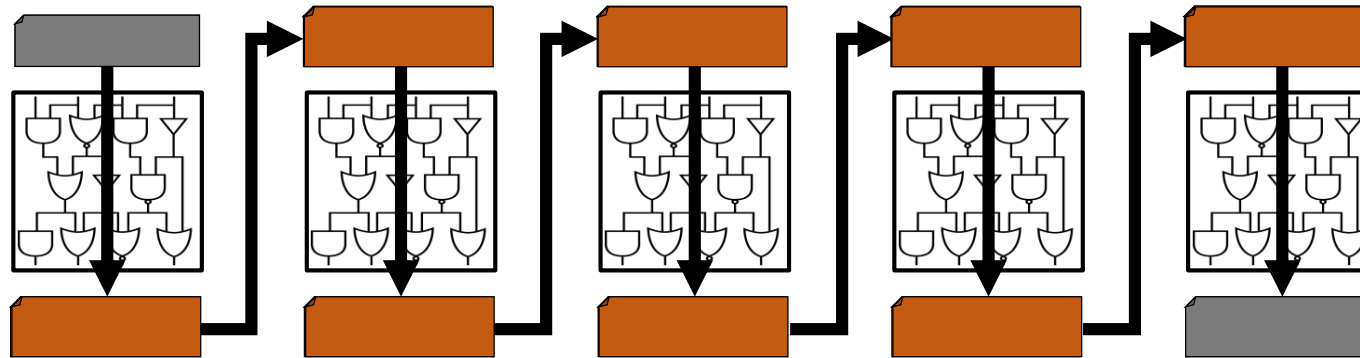
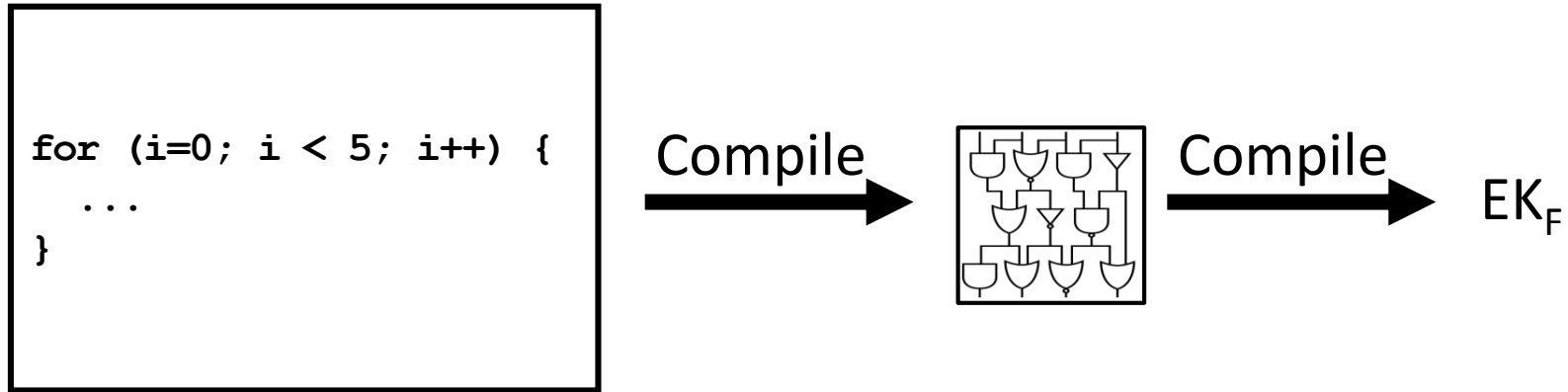
Verifying Stateful Computations: Loops



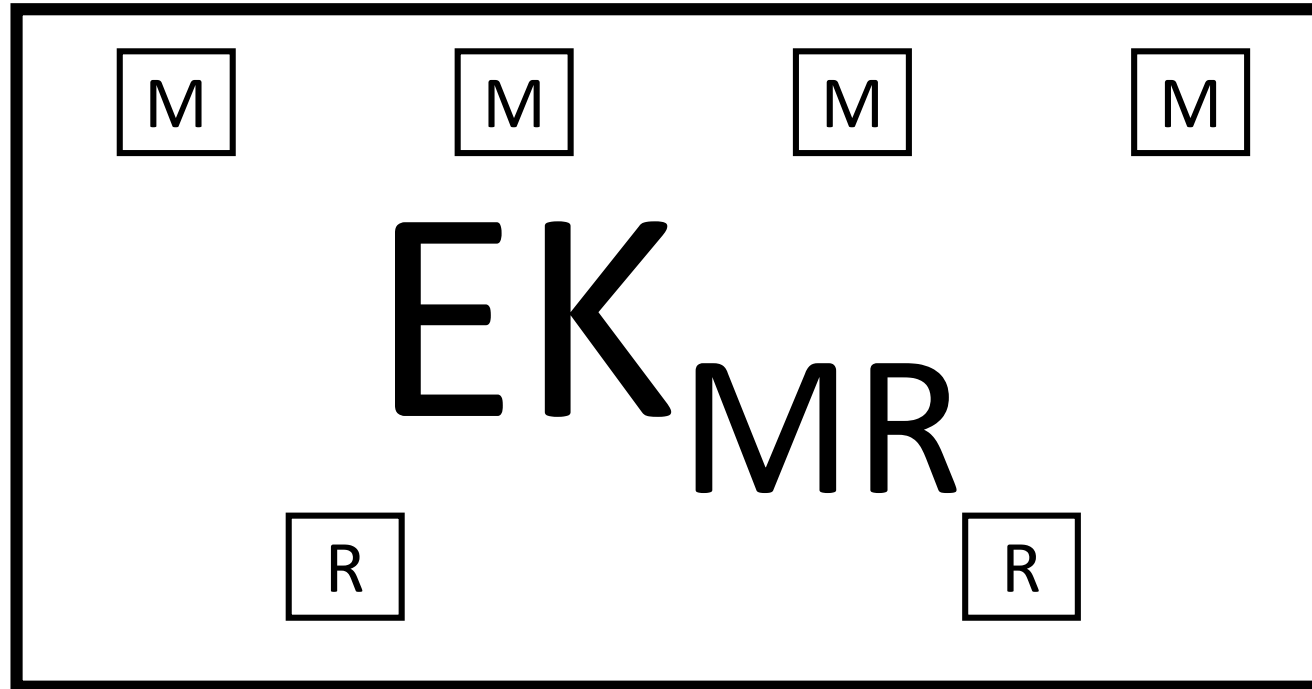
Verifying Stateful Computations: Loops



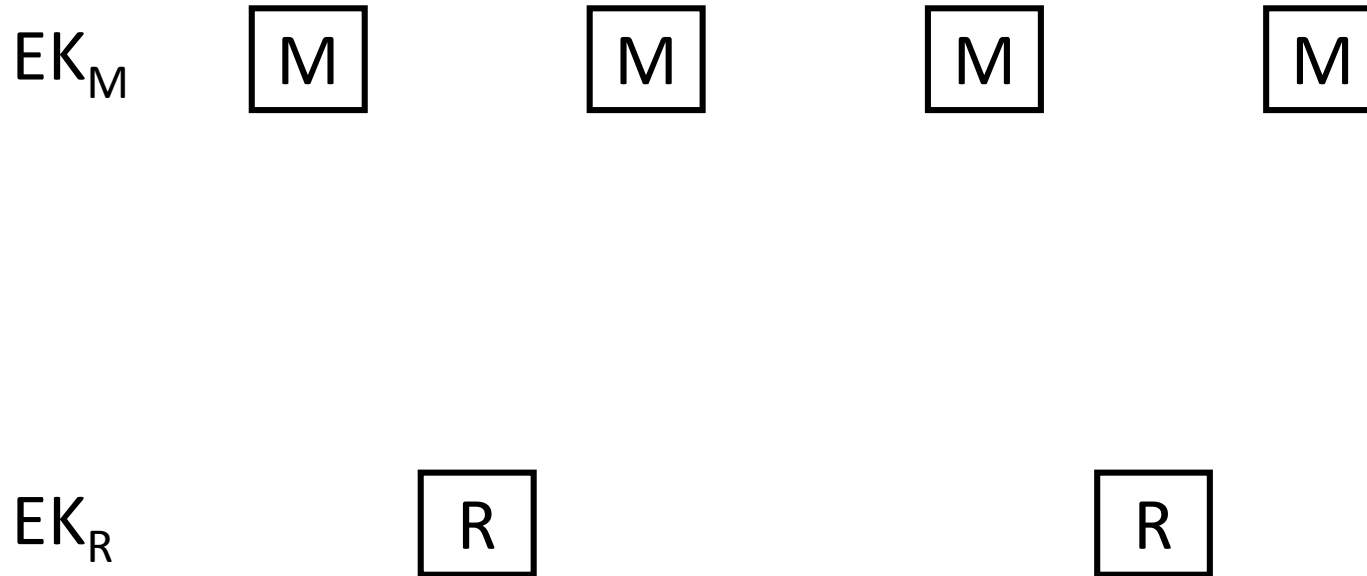
Verifying Stateful Computations: Loops



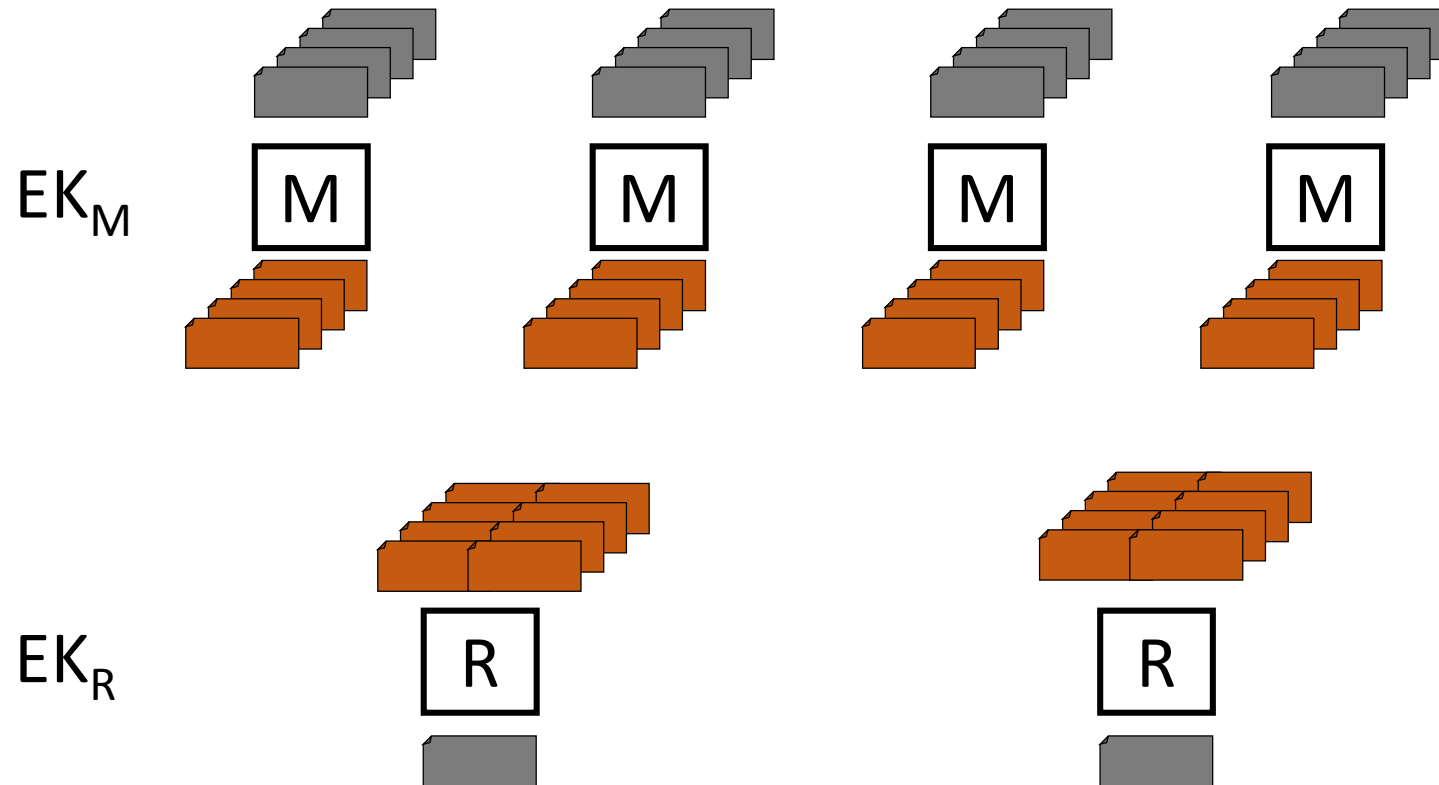
Verifying Stateful Computations: MapReduce



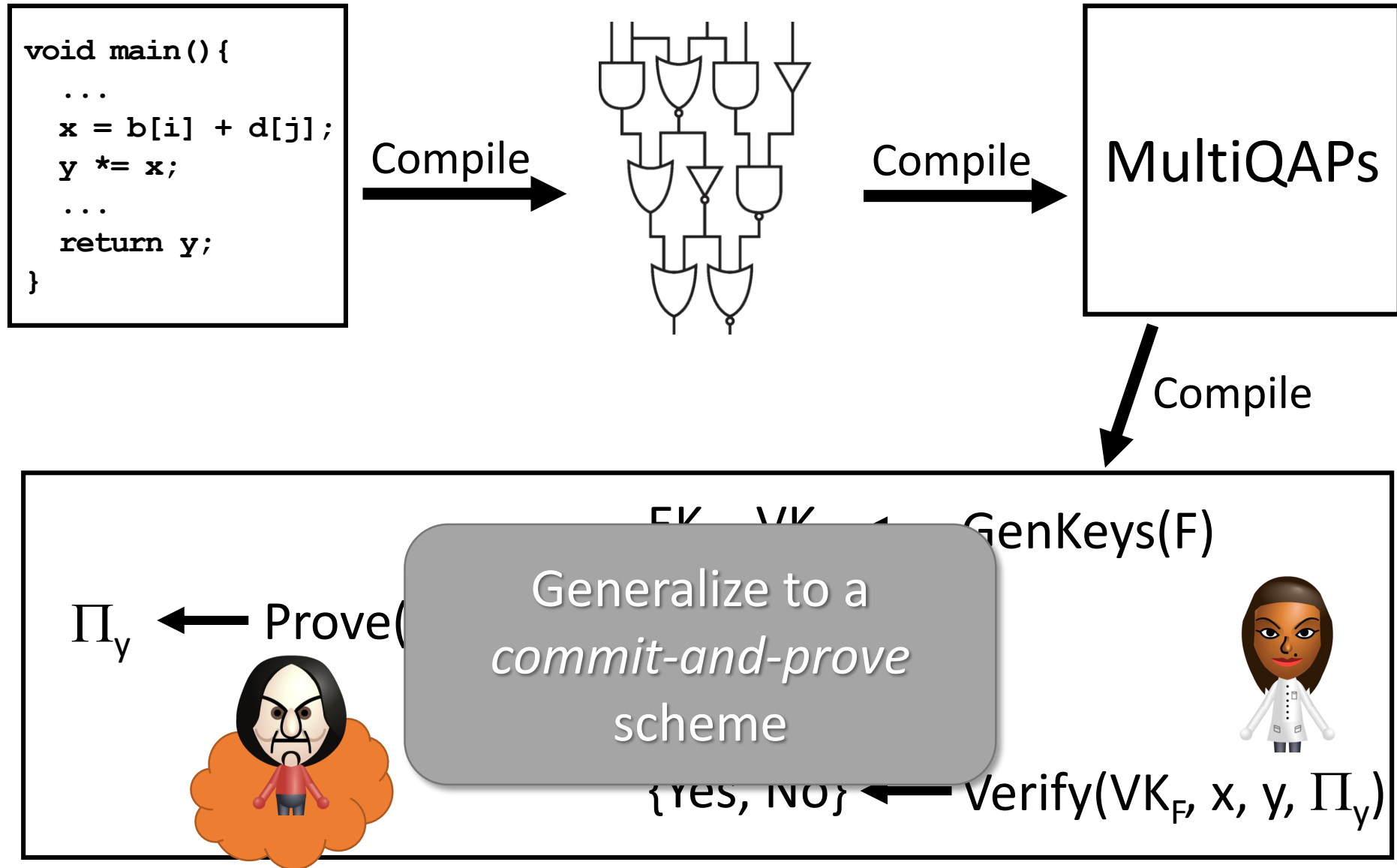
Verifying Stateful Computations: MapReduce



Verifying Stateful Computations: MapReduce



Stateful Computations with Geppetto



Geppetto's Commit-and-Prove Scheme

$EK_F, VK_F \leftarrow \text{GenKeys}(F)$

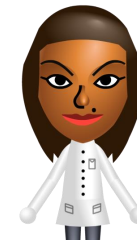
$C_x \leftarrow \text{Commit}(EK_F, \vec{x})$

$\Pi_F \leftarrow \text{Prove}(EK_F, \{C_1, \dots, C_N\})$

$\Pi_y \leftarrow \text{Prove}(EK_F, x, y)$



$\{\text{Yes}, \text{No}\} \leftarrow \text{VerifyCommit}(VK_F, C_x)$



$\{\text{Yes}, \text{No}\} \leftarrow \text{VerifyProof}(VK_F, \Pi_F, \Pi_y)$

Core Geppetto's scheme



$$\text{KeyGen}(R) \longrightarrow EK, VK$$

Generate the MultiQAP for R

Pick random s

Compute $EK = \{EK_j\}, \{g^{s^i}\}$

$$EK_j = \left\{ \begin{array}{l} g^{v_k(s)}, g^{w_k(s)}, g^{y_k(s)} \\ g^{\alpha_{j,v}v_k(s)}, g^{\alpha_{j,w}w_k(s)}, g^{\alpha_{j,y}y_k(s)} \end{array} \right\}_{k \in I_j}$$

Compute $VK = (g^{d(s)} = g^{\prod_i (s-i)})$

$$\text{Verify}(VK_j, C_j)$$

$$e(g^{v(j)}, g_j^{\alpha_{j,v}}) = e(g^{\alpha_{j,v}v(j)}, g)$$

and similarly for w and y



$$\text{Commit}(EK_j, u_j, o_j) \longrightarrow C_j$$

Generate the commitment:

$$v^{(j)}(s) = \sum_{k \in I_j} u_k v_k(s) + o_{j,w} d(s), \text{ similarly for } w \text{ and } y$$

$$C_j = (g^{v(j)}, g^{w(j)}, g^{y(j)}, g^{\alpha_{j,v}v(j)}, g^{\alpha_{j,w}w(j)}, g^{\alpha_{j,y}y(j)}).$$

$v(s) = \prod v^{(j)}$ and similarly for w and y

$$\text{Prove}(EK, \mathbf{u}, \mathbf{o}) \longrightarrow \pi$$

Find $h(x)$ s.t. $h(x) * d(x) = v(x) * w(x) - y(x)$

Compute $g^{h(s)} = \prod (g^{s^i})^{h_i}$

Proof is $(g^{v(s)}, g^{w(s)}, g^{y(s)}, g^{h(s)})$

Prove/Verify this polynomial equality for a random secret s ; then use Schwartz-Zippel Lemma

$$\text{Verify}(VK, \mathbf{C}, \pi) \longrightarrow \{\text{Yes}, \text{No}\}$$

$$\frac{e(g^{v(s)}, g^{w(s)})}{e(g^{y(s)}, g)} \stackrel{?}{=} e(g^{h(s)}, g^{d(s)}) \quad \left. \vphantom{\frac{e(g^{v(s)}, g^{w(s)})}{e(g^{y(s)}, g)} \stackrel{?}{=} e(g^{h(s)}, g^{d(s)})} \right\} e(\cdot, \cdot) \text{ is a pairing: } e(g^p, g^q) = e(g, g)^{pq}$$

Commit & Prove: security

- Geppetto commitments are **binding** under the q-SDH assumption
- Geppetto commitments and proofs are **perfectly hiding**.
- Geppetto is **knowledge-sound** under prior q-PKE, q-SDH, and q-DHE assumptions and is **perfectly zero-knowledge**.

q-PKE:

$$\begin{aligned}
 Pr[& \mathcal{G}_\lambda \leftarrow \mathcal{G}(1^\lambda); g \leftarrow \mathbb{G}_1^* \times \mathbb{G}_2^*; \alpha, s \leftarrow \mathbb{Z}_p^*; \\
 & \sigma \leftarrow (\mathcal{G}_\lambda, g, g^s, \dots, g^{s^q}, g^\alpha, g^{\alpha s}, \dots, g^{\alpha s^q}) \\
 & z \leftarrow \mathcal{S}(\mathcal{G}_\lambda, g, g^s, \dots, g^{s^q}); \\
 & (c, \hat{c}; a_0, \dots, a_q) \leftarrow (\mathcal{A} \parallel \chi_{\mathcal{A}})(\sigma, z) : \\
 & \hat{c} = c^\alpha \wedge c \neq \prod_{i=0}^q g_1^{a_i s^i}] = \text{negl}(\lambda)
 \end{aligned}$$

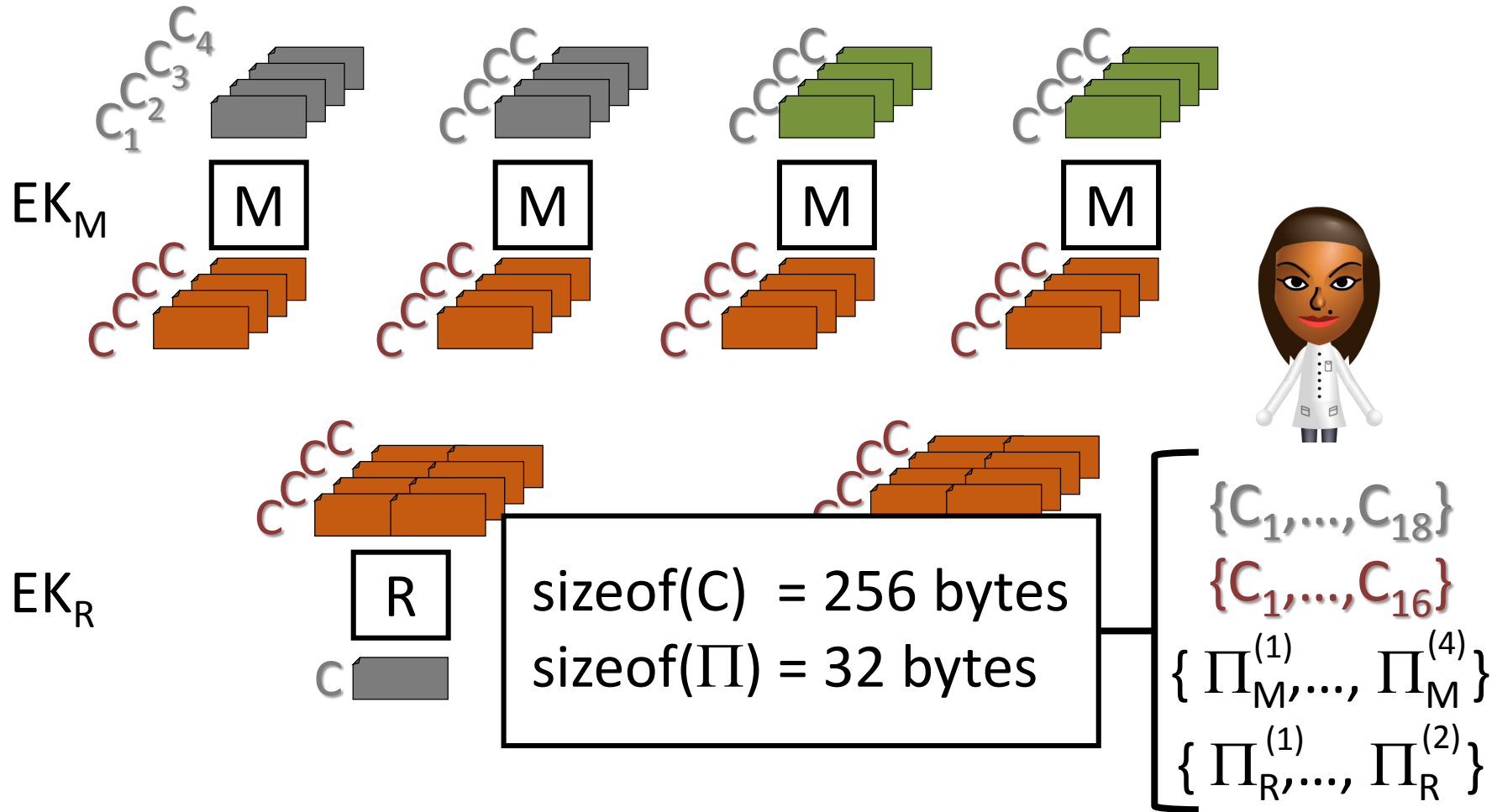
q-SDH:

$$\begin{aligned}
 Pr[& (\mathcal{G}_\lambda \leftarrow \mathcal{G}(1^\lambda); g \leftarrow \mathbb{G}_1^* \times \mathbb{G}_2^*; s \leftarrow \mathbb{Z}_p^*; \sigma \leftarrow (\mathcal{G}_\lambda, g, g^s, \dots, g^{s^q}); \\
 & y \leftarrow \mathcal{A}(\sigma) : y = e(g_1, g_2)^{\frac{1}{s+c}}, c \in \mathbb{Z}_p^*] = \text{negl}(\lambda).
 \end{aligned}$$

q-DHE:

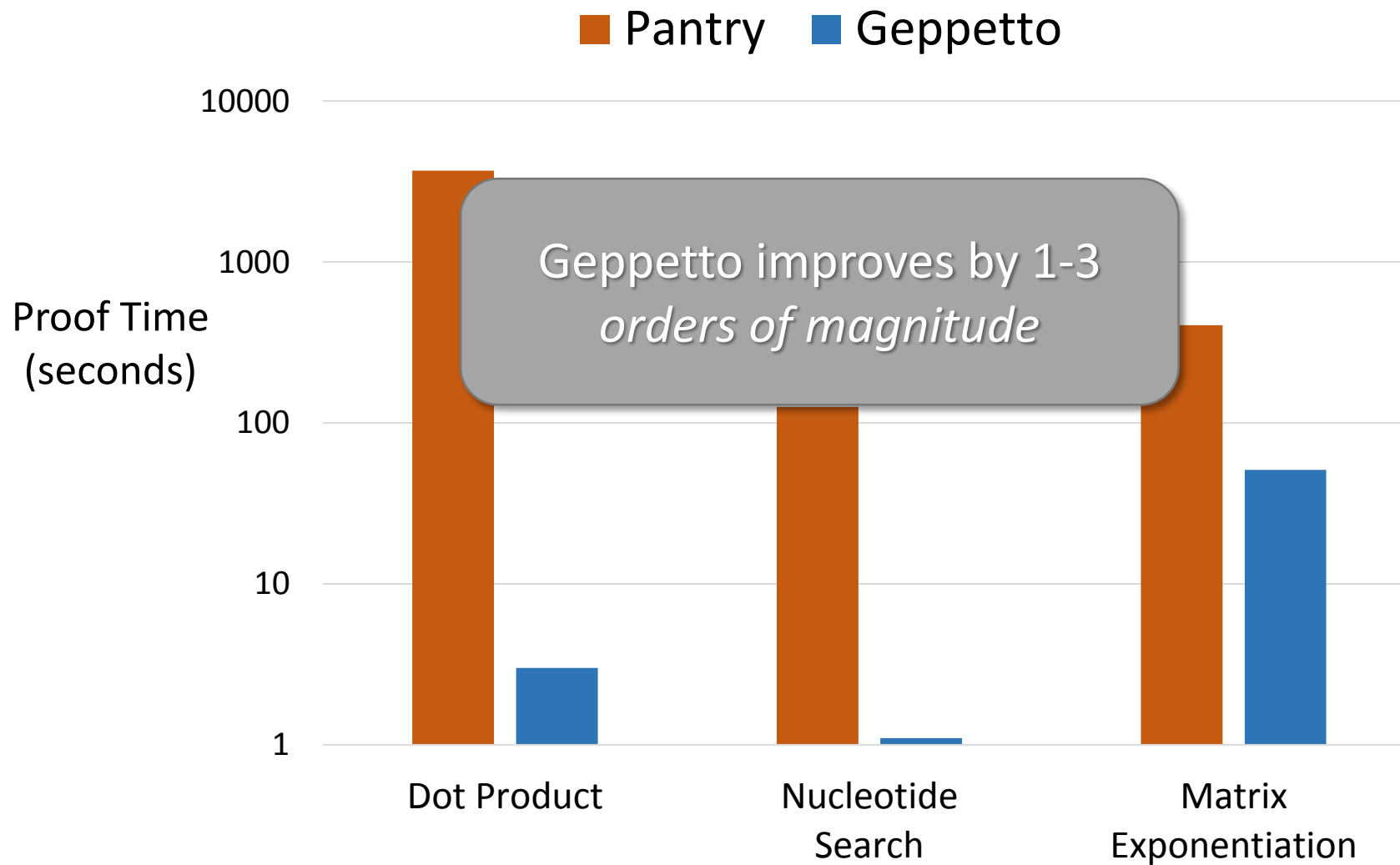
$$\begin{aligned}
 Pr[& \mathcal{G}_\lambda \leftarrow \mathcal{G}(1^\lambda); g \leftarrow \mathbb{G}_1^* \times \mathbb{G}_2^*; s \leftarrow \mathbb{Z}_p^*; \\
 & \sigma \leftarrow (\mathcal{G}_\lambda, g, g^s, \dots, g^{s^q}, g^{s^{q+2}}, \dots, g^{s^{2q}}); \\
 & y \leftarrow \mathcal{A}(\sigma) : y = g_1^{s^{q+1}}] = \text{negl}(\lambda).
 \end{aligned}$$

MapReduce with Geppetto

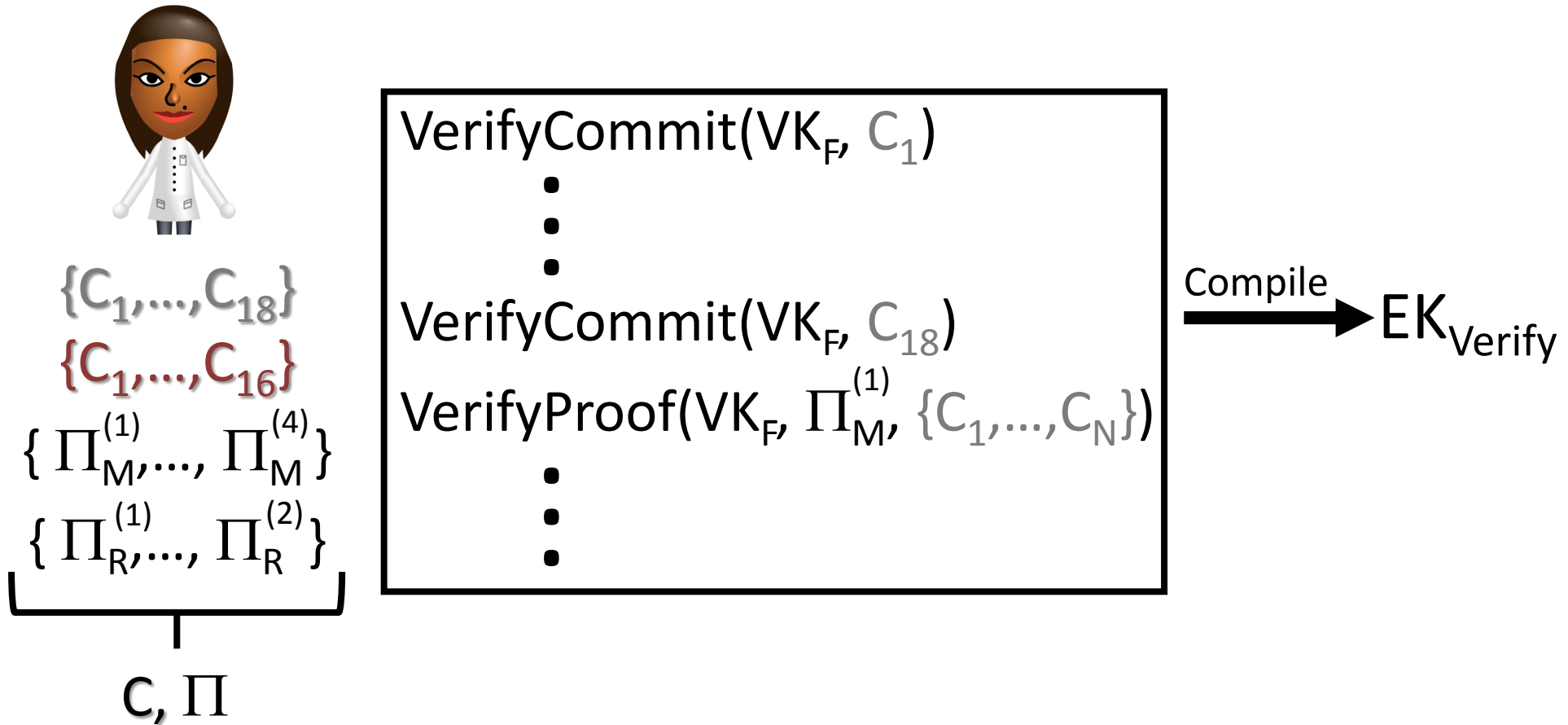


Evaluation: Geppetto vs. Pantry

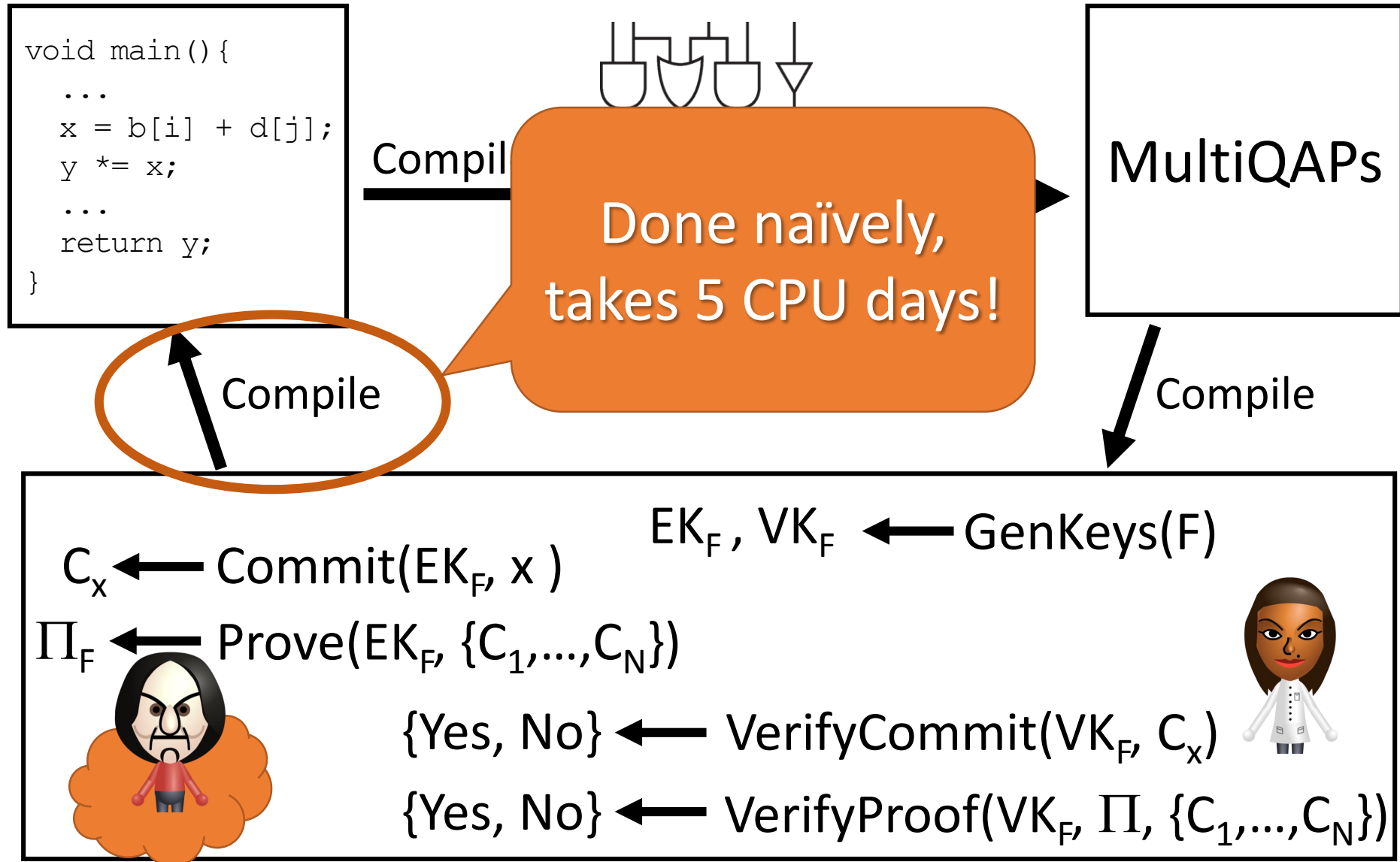
[Braun et al. SOSP '14]



Verified Bootstrapping [Valiant '08], [Bitansky et al. '13]

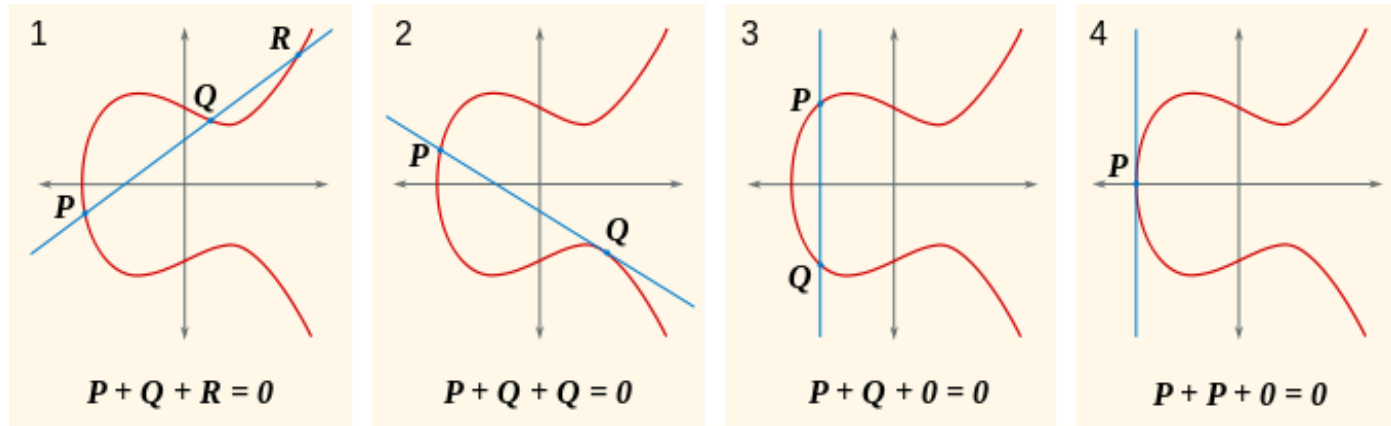


Verified Bootstrapping



Towering Elliptic Curves

Points (x, y) such that $y^2 = x^3 + ax + b$.



- We use a curve in F_p to prove QAPs over F_p ,
(using point addition for addition & pairing for multiplication)
- Naively compiling a C implementation of pairing
is exorbitantly expensive, $\sim 44 M^*$ for each pairing.
- For bootstrapping,
we use a curve in $F_{\tilde{p}}$ to prove QAPs over F_p
and we adjust our implementation

Bootstrapping using Cycles of Elliptic Curves

Scalable Zero Knowledge via Cycles of Elliptic Curves

Eli Ben-Sasson¹, Alessandro Chiesa², Eran Tromer³, and Madars Virza²

¹ Technion, Haifa, Israel

eli@cs.technion.ac.il

² MIT, Cambridge, MA, USA

{alexch,madars}@csail.mit.edu

³ Tel Aviv University, Tel Aviv, Israel

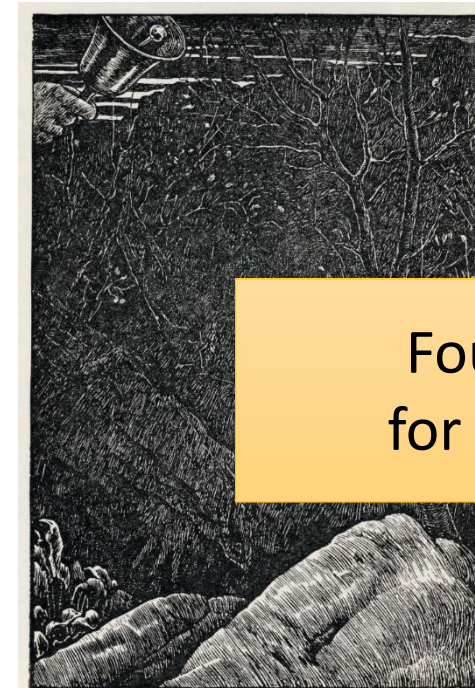
tromer@cs.tau.ac.il

Abstract. Non-interactive zero-knowledge proofs of knowledge for general NP statements are a powerful cryptographic primitive, both in theory and in practical applications. Recently, much research has focused on achieving an additional property, *succinctness*, requiring the proof to be very short and easy to verify. Such proof systems are known as *zero-knowledge succinct non-interactive arguments of knowledge* (zk-SNARKs), and are desired when communication is expensive, or the verifier is computationally weak.

Existing zk-SNARK implementations have severe scalability limitations, in terms of space complexity as a function of the size of the computation being proved (e.g., running time of the NP statement's decision program). First, the size of the proving key is quasilinear in the upper bound on the computation size. Second, producing a proof requires “writing down” all intermediate values of the entire computation, and then conducting global operations such as FFTs.

The bootstrapping technique of Bitansky et al. (STOC '13), following Valiant (TCC '08), offers an approach to scalability, by recursively composing proofs: proving statements about acceptance of the proof system's own verifier (and correctness of the program's latest step). Alas, recursive composition of known zk-SNARKs has never been realized in practice, due to enormous computational cost.

Using new elliptic-curve cryptographic techniques, and methods for exploiting the proof systems' field structure and nondeterminism, we achieve the first zk-SNARK implementation that practically achieves recursive proof composition. Our zk-SNARK implementation runs random-access machine programs and produces proofs of their correct execution, on today's hardware, for any pro-



Found 2 curves
for which $\tilde{p} = p'$

Bootstrapping using the Cocks-Pinch Method

Lemma. Let $x \in \mathbb{Z}$ be such that both $p = 36x^4 + 36x^3 + 24x^2 + 6x + 1$ and $p' = 36x^4 + 36x^3 + 18x^2 + 6x + 1$ are prime. If

$$\tilde{p} = 5184x^8 + 10368x^7 + 12204x^6 + 8856x^5 + 4536x^4 + 1548x^3 + 363x^2 + 48x + 4$$

is also prime, there exist

- a curve E/F_p of order $p' = \#E(F_p)$ with embedding degree 12 (w.r.t. p')
- a curve $\tilde{E}/F_{\tilde{p}}$ of order $\#\tilde{E}(F_{\tilde{p}})$, a multiple of p , with embedding degree 6 (w.r.t. p).

General vs Bounded Bootstrapping

Ben-Sasson et al. [CRYPTO 2014] **scalable, but not yet practical**

- Brute force search of curve:
only 80bit security level
- Use MNT curves, not most efficient
at high security levels
- Overhead of at least 34-77x for
exponentiation and 11x for pairing
- Interpretation (mHz)

Cocks-Pinch method **nearly practical, but restricted**

- General formula for finding curves
- Can use BN curves for inner curve
- Support curves for different
security levels (e.g. 128bit security)
- Bounded bootstrapping
- Compilation (kHz)

Evaluation: Bootstrapping

Prior Work

[Ben-Sasson et al. '14]

Unbounded
recursion

Q
in

Geppetto improves by up to
5 orders of magnitude

29 *years*

Geppetto

Fixed bound

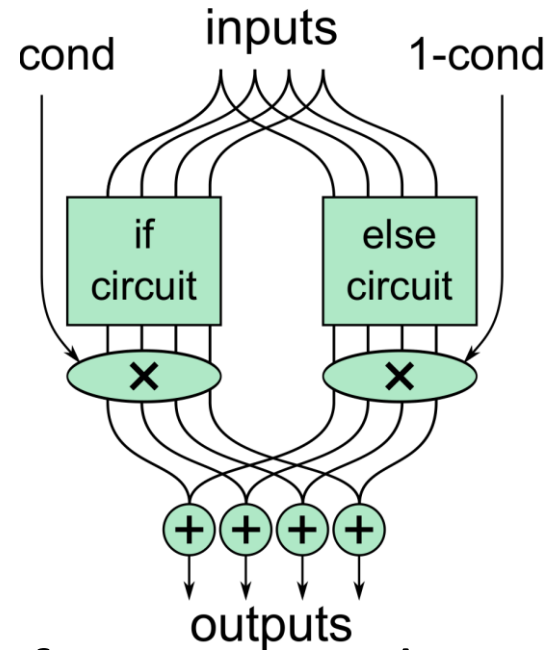
152 *minutes*

Energy-Saving Compilation

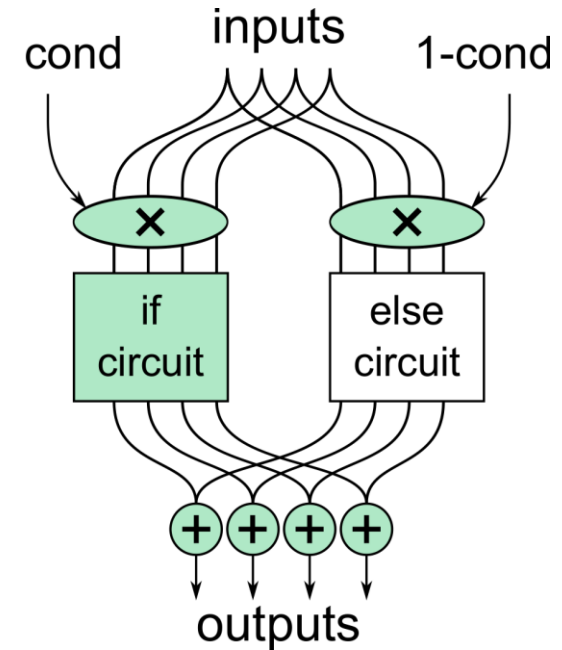
```

if (cond) {
    ...
} else {
    ...
}
    
```

Standard



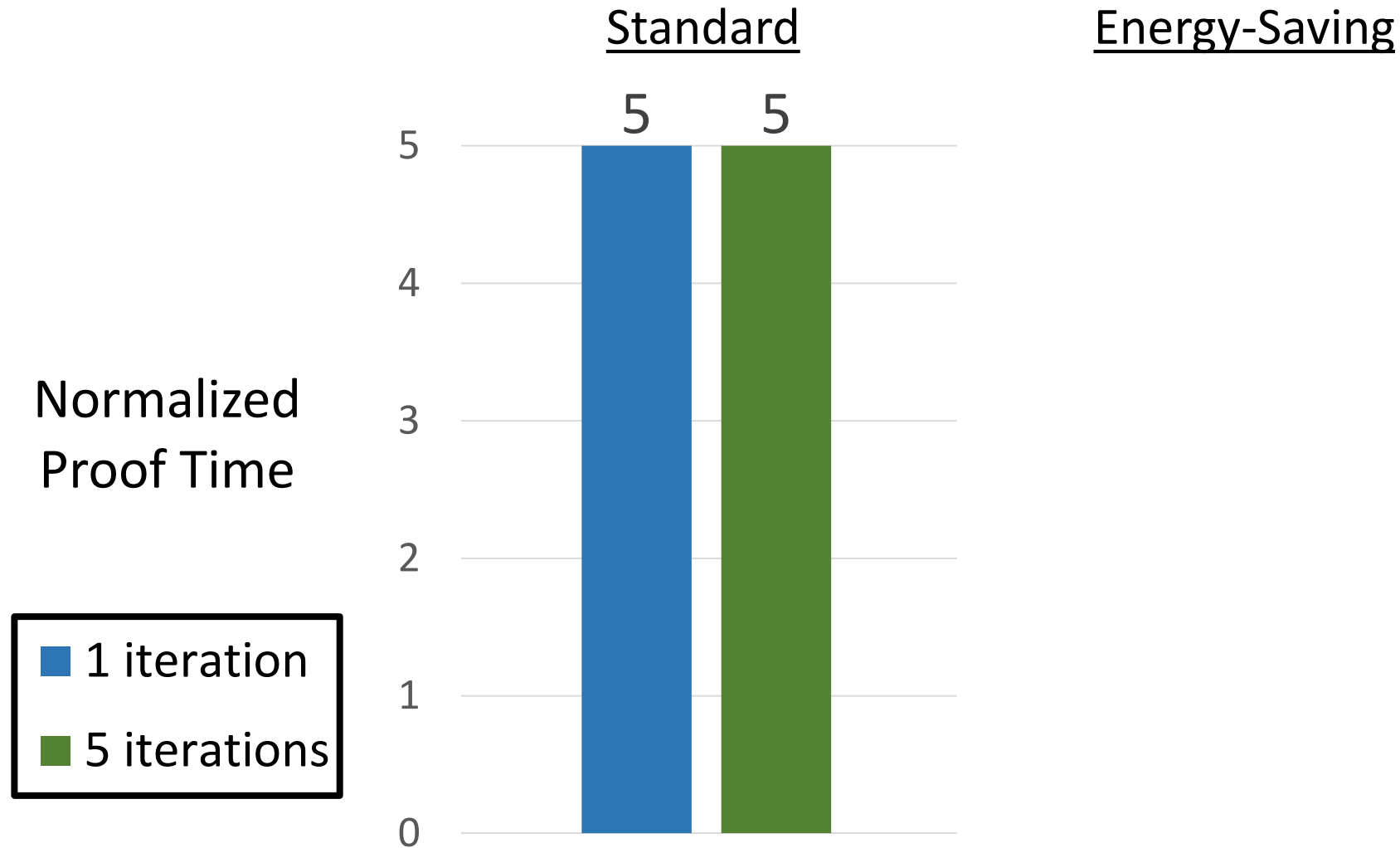
Energy-Saving



$C \xleftarrow{\text{Commit}} \overbrace{g^{W1} \dots g^{Wk}}^{\text{if}} \overbrace{g^{Wk+1} \dots g^{Wn}}^{\text{else}}$

$g^{W1} \dots g^{Wk} \underbrace{g^0 \dots g^0}_{\text{Energy-Saving}}$

Energy-Saving Compilation



Is Geppetto correct?

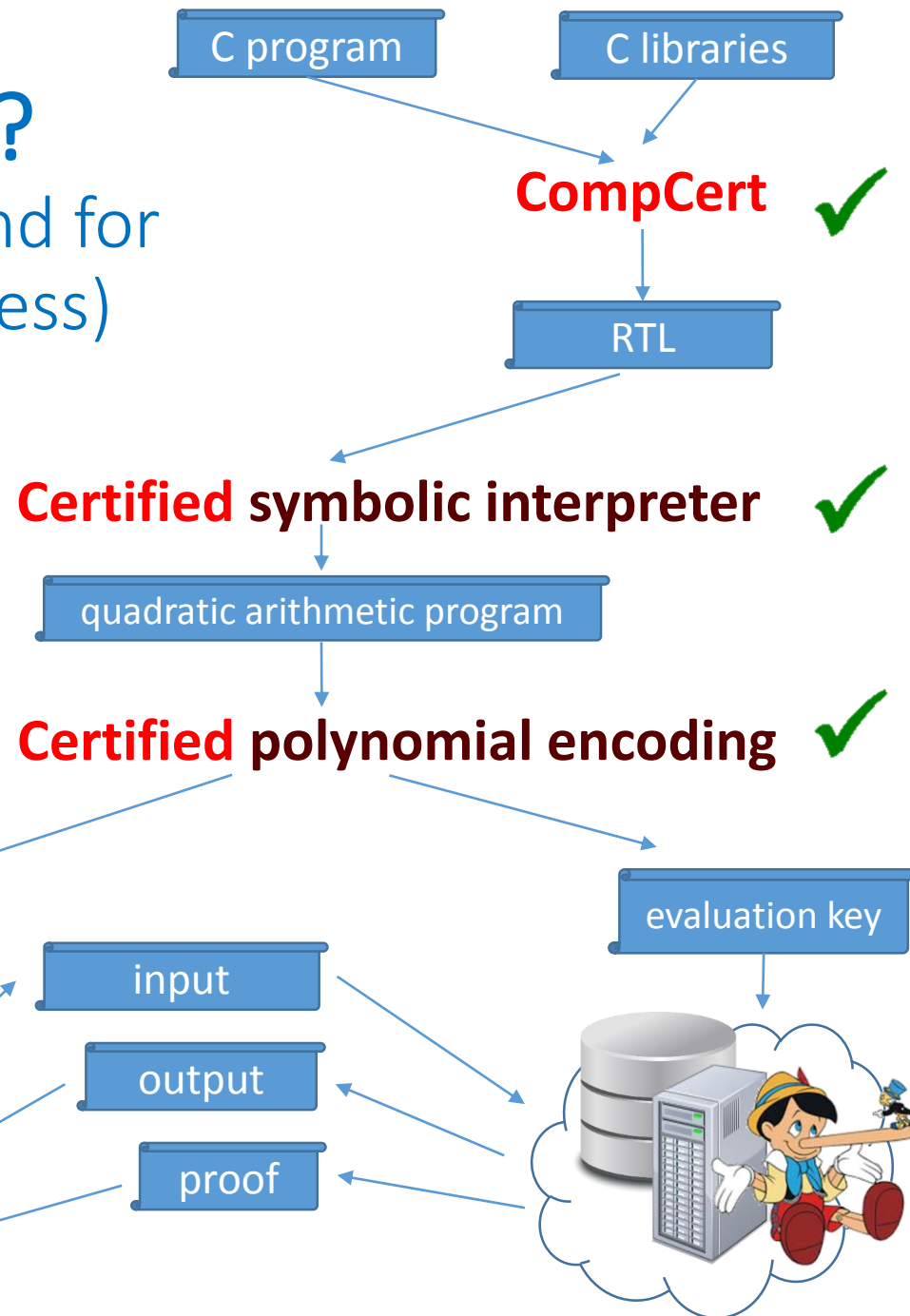
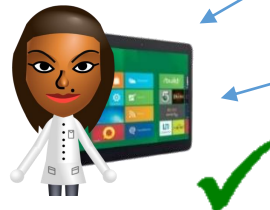
a certified crypto back-end for
CompCert (work in progress)

The compiler

(extracted from Coq)

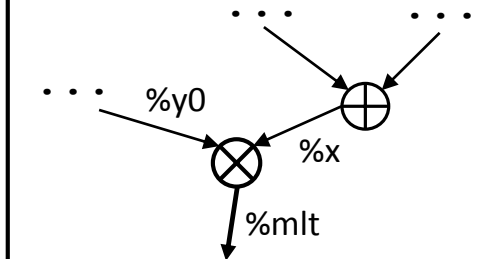
- Generates QAPs
- Ensures that, if some polynomial equation holds, then the source program has a trace with matching I/O.

The crypto verifier ensures
that this equation holds
(with probability $1 - \epsilon$)



```
void main() {  
    ...  
    x = b[i] + d[j];  
    y *= x;  
    return y; }
```

```
define i32 @main() #1 {  
entry:  
    ...  
    %y0 = load i32* %y.addr  
    %mlt = mul nsw i32 %x, %y0  
    ret i32 %mlt }
```

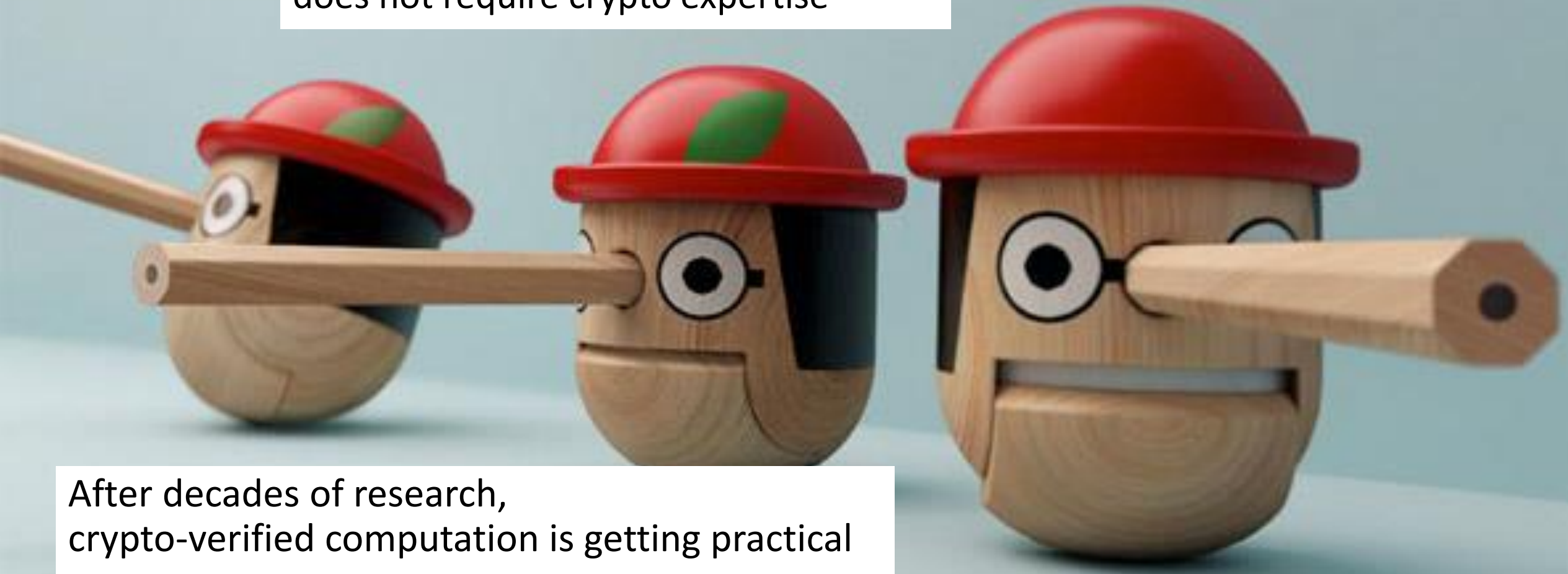


```
bc96 a736 6e97 8bfc 957f 5857 d4a6 2770  
f3cb f2e0 d5f5 de09 f46a 7d00 e8db 0e0a  
7430 a958 9875 9d7a 5d5e 0cec 955d 13c5  
b605 be6d 1685 72fa c3e9 dc09 c7dc cc05  
ca47 04cc e739 663f 3c4a 9dea c74b 7c5b
```

Geppetto: **Versatile Verifiable Computation**

Geppetto compiles useful C programs,
does not require crypto expertise

Early applications
verify computations
on private data



After decades of research,
crypto-verified computation is getting practical