Verifiable ASICs

Riad S. Wahby[⋆◦] rsw@cs.stanford.edu
Max Howald[†] howald@cooper.edu
Siddharth Garg[⋆] sg175@nyu.edu
abhi shelat[‡] abhi@virginia.edu
Michael Walfish[⋆] mwalfish@cs.nyu.edu

[⋆]New York University [◦]Stanford University [†]The Cooper Union [‡]The University of Virginia

***Abstract*—A manufacturer of custom hardware (ASICs) can undermine the intended execution of that hardware; high-assurance execution thus requires controlling the manufacturing chain. However, a trusted platform might be orders of magnitude worse in performance or price than an advanced, untrusted platform. This paper initiates exploration of an alternative: using verifiable computation (VC), an untrusted ASIC computes *proofs* of correct execution, which are verified by a trusted processor or ASIC. In contrast to the usual VC setup, here the prover and verifier *together* must impose less overhead than the alternative of executing directly on the trusted platform. We instantiate this approach by designing and implementing physically realizable, area-efficient, high throughput ASICs (for a prover and verifier), in fully synthesizable Verilog. The system, called Zebra, is based on the CMT and Allspice interactive proof protocols, and required new observations about CMT, careful hardware design, and attention to architectural challenges. For a class of real computations, Zebra meets or exceeds the performance of executing directly on the trusted platform.**

1 Introduction

When the designer of an *ASIC* (application specific integrated circuit, a term that refers to custom hardware) and the manufacturer of that ASIC, known as a *fab* or *foundry*, are separate entities, the foundry can mount a *hardware Trojan* [19, 35, 105] attack by including malware inside the ASIC. Government agencies and semiconductor vendors have long regarded this threat as a core strategic concern [4, 10, 15, 18, 45, 81, 111].

The most natural response—achieving high assurance by controlling the manufacturing process—may be infeasible or impose enormous penalties in price and performance.[1] Right now, there are only five nations with top-end foundries [69] and only 13 foundries among them; anecdotally, only four foundries will be able to manufacture at 14 nm or beyond. In fact, many advanced nations do not have any onshore foundries. Others have foundries that are generations old; India, for example, has 800nm technology [12], which is 25 years older and $10^8\times$ worse (when considering the product of ASIC area and energy) than the state of the art.

Other responses to hardware Trojans [35] include post-fab detection [19, 26, 73, 77, 78, 119] (for example, testing based on input patterns [47, 120]), run-time detection and disabling (for example, power cycling [115]), and design-time obfuscation [46, 70, 92]. These techniques provide some assurance under certain misbehaviors or defects, but they are not sensitive enough to defend against a truly adversarial foundry (§10).

One may also apply N-versioning [49]: use two foundries, deploy the two ASICs together, and, if their outputs differ, a trusted processor can act (notify an operator, impose fail-safe behavior, etc.). This technique provides no assurance if the foundries collude—a distinct possibility, given the small number of top-end fabs. A high assurance variant is to execute the desired functionality in software or hardware on a trusted platform, treating the original ASIC as an untrusted accelerator whose outputs are checked, potentially with some lag.

This leads to our motivating question: *can we get high-assurance execution at a lower price and higher performance than executing the desired functionality on a trusted platform?* To that end, this paper initiates the exploration of *verifiable ASICs* (§2.1): systems in which deployed ASICs prove, each time they perform a computation, that the execution is correct, in the sense of matching the intended computation.[2] An ASIC in this role is called a *prover*; its proofs are efficiently checked by a processor or another ASIC, known as a *verifier*, that is trusted (say, produced by a foundry in the same trust domain as the designer). The hope is that this arrangement would yield a positive response to the question above. But is the hope well-founded?

On the one hand, this arrangement roughly matches the setups in probabilistic proofs from complexity theory and cryptography: interactive proofs or IPs [22, 63, 64, 80, 102], efficient arguments [39, 72, 74, 83], SNARGs [62], SNARKs [36], and verifiable outsourced computation [23, 60, 63, 83] all yield proofs of correct execution that can be efficiently checked by a verifier. Moreover, there is a flourishing literature surrounding the refinement and implementation of these protocols [24, 25, 29, 31–33, 41, 51, 55, 56, 58, 59, 61, 76, 88, 98–101, 106, 108, 112, 114] (see [117] for a survey). On the other hand, all of this work can be interpreted as a *negative* result: despite impressive speedups, the resulting artifacts are not deployable for the application of verifiable offloading. The biggest problem is the prover's burden: its computational overhead is at least $10^5\times$, and usually more than $10^7\times$, greater than the cost of just executing the computation [117, Fig. 5].

Nevertheless, this issue is potentially surmountable—at least in the hardware context. With CMOS technology, many costs scale down super-linearly; as examples, area and energy reduce with the square and cube of critical dimension, respectively [91]. As a consequence, the performance improvement, when going from an ASIC manufactured in a trusted, older foundry to one manufactured in an untrusted, advanced

[1] Creating a top-end foundry requires both rare expertise and billions of dollars, to purchase high-precision equipment for nanometer-scale patterning and etching [52]. Furthermore, these costs worsen as the *technology node*—the manufacturing process, which is characterized by the length of the smallest transistor that can be fabricated, known as the *critical dimension*—improves.

[2] This is different from, but complementary to, the vast literature on hardware verification. There, the goal is to statically verify that a circuit design—which is assumed to be manufactured faithfully—meets an intended specification.

foundry, can be *larger* than the overhead of provers in the aforementioned systems (as with the earlier example of India).

Given this gap, verifiable outsourcing protocols could yield a positive answer to our motivating question—but only if the prover can be implemented on an ASIC in the first place. And that is easier said than done. Many protocols for verifiable outsourcing [24, 25, 29, 31–33, 41, 51, 56, 58, 59, 61, 76, 88, 99–101, 114] have concrete bottlenecks (cryptographic operations, serial phases, communication patterns that lack temporal and spatial locality, etc.) that seem incompatible with an efficient, physically realizable hardware design. (We learned this the hard way; see Section 9.)

Fortunately, there is a protocol in which the prover uses no cryptographic operations, has highly structured and parallel data flows, and demonstrates excellent spatial and temporal locality. This is CMT [55], an interactive proof that refines GKR [63]. Like all implemented protocols for verifiable outsourcing, CMT works over computations expressed as *arithmetic circuits*, meaning, loosely speaking, additions and multiplications. To be clear, CMT has further restrictions. As enhanced by Allspice [112], CMT works best for computations that have a parallel and numerical flavor and make sparing use of non-arithmetic operations (§2.2). But we can live with these restrictions because there are computations that have the required form (the number theoretic transform, polynomial evaluation, elliptic curve operations, pattern matching with don't cares, etc.; see also [55, 88, 99–101, 106, 108, 112]).

Moreover, one might expect something to be sacrificed, since our setting introduces additional challenges to verifiable computation. First, working with hardware is inherently difficult. Second, whereas the performance requirement up until now has been that the verifier save work versus carrying out the computation directly [41, 55, 60, 88, 99–101, 112, 114], here we have the additional requirement that the *whole system*—verifier together with prover—has to beat that baseline.

This brings us to the work of this paper. We design, implement, and evaluate physically realizable, high-throughput ASICs for a prover and verifier based on CMT and Allspice. The overall system is called *Zebra*.

Zebra's design (§3) is based, first, on new observations about CMT, which yield parallelism (beyond that of prior work [108]). One level down, Zebra arranges for locality and predictable data flow. At the lowest level, Zebra's design is latency insensitive [44], with few timing constraints, and it reuses circuitry to reduce ASIC area. Zebra also responds to architectural challenges (§4), including how to meet the requirement, which exists in most implemented protocols for verifiable computation, of trusted offline precomputation; how to endow the verifier with storage without the cost of a trusted storage substrate; and how to limit the overhead of the verifier-prover communication.

The core design is fully implemented (§6): Zebra includes a compiler that takes an arithmetic circuit description to synthesizable Verilog (meaning that the Verilog can be compiled to a hardware implementation). Combined with existing compilers that take C code to arithmetic circuit descriptions [31–33, 40, 41, 51, 56, 88, 99, 101, 112, 114], Zebra obtains a pipeline in which a human writes high-level software, and a toolchain produces a hardware design.

Our evaluation of Zebra is based on detailed modeling (§5) and measurement (§7). Taking into account energy, area, and throughput, Zebra outperforms the baseline when both of the following hold: (a) the technology gap between $\mathscr{P}$ and $\mathscr{V}$ is a more than a decade, and (b) the computation of interest can be expressed naturally as an arithmetic circuit with tens of thousands of operations. An example is the number theoretic transform (§8.1): for 2^{10}-point transforms, Zebra is competitive with the baseline; on larger computations it is better by 2–3×. Another example is elliptic curve point multiplication (§8.2): when executing several hundred in parallel, Zebra outperforms the baseline by about 2×.

Zebra has clear limitations (§9). Even in the narrowly defined regime where it beats the baseline, the price of verifiability is very high, compared to untrusted execution. Also, Zebra has some heterodox manufacturing and operating requirements (§4); for example, the operator must periodically take delivery of a preloaded hard drive. Finally, Zebra does not have certain properties that other verifiable computation do: low round complexity, public verifiability, zero knowledge properties, etc. On the other hand, these amenities aren't needed in our context.

Despite the qualified results, we believe that this work, viewed as a first step, makes contributions to hardware security and to verifiable computation:

- It initiates the study of verifiable ASICs. The high-level notion had been folklore (for example, [50, §1.1]), but there have been many details to work through (§2.1, §2.3).
- It demonstrates a response to hardware Trojans that works in a much stronger threat model than prior defenses (§10).
- It makes new observations about CMT. While they are of mild theoretical interest (at best), they matter a lot for the efficiency of an implementation.
- It includes a hardware design that achieves efficiency by composing techniques at multiple levels of abstraction. Though their novelty is limited (in some cases, it is none), together they produce a milestone: the first hardware design and implementation of a probabilistic proof system.
- It performs careful modeling, accounting, and measurement. At a higher level, this is the first work to identify a setting in which one can simultaneously capture the "cost" of a prover and verifier together, and to give implementations for which this quantity is (sometimes) less expensive than having the verifier compute on its own.

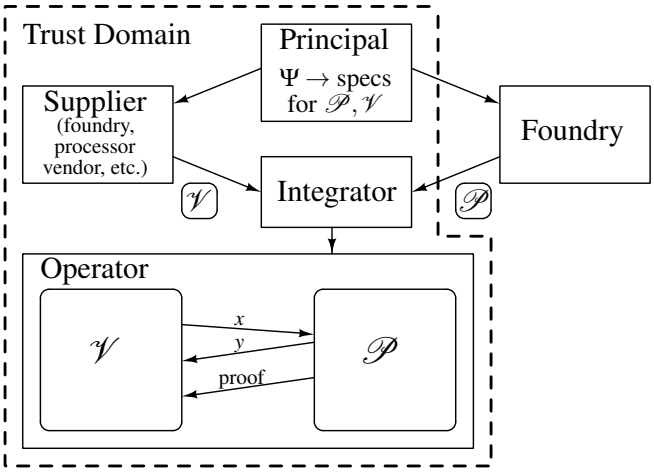


FIGURE 1—Verifiable ASICs. A principal outsources the production of an ASIC ($\mathcal{P}$) to an untrusted foundry and gains high-assurance execution via a trusted verifier ($\mathcal{V}$) and a probabilistic proof protocol.

2 Setup and starting point

2.1 Problem statement: verifiable ASICs

The setting for *verifiable ASICs* is depicted in Figure 1. There is a **principal**, who defines a *trust domain*. The principal could be a government, a fabless semiconductor company that designs circuits, etc. The principal wishes to deploy an ASIC that performs some computation Ψ, and wishes for a third party—outside of the trust domain and using a state-of-the-art technology node (§1)—to manufacture the chip. After fabrication, the ASIC, which we call a *prover* $\mathcal{P}$, is delivered to and remains within the trust domain. In particular, there is a trusted step in which an *integrator*, acting on behalf of the principal, produces a single system by combining $\mathcal{P}$ with a *verifier* $\mathcal{V}$. The operator or end user trusts the system that the integrator delivers to it.

The manufacturer or supplier of $\mathcal{V}$ is trusted by the principal. For example, $\mathcal{V}$ could be an ASIC manufactured at a less advanced but trusted foundry located onshore [15, 45]. Or $\mathcal{V}$ could run in software on a general-purpose CPU manufactured at such a foundry, or on an existing CPU that is assumed to have been manufactured before Trojans became a concern. We refer to the technology node of $\mathcal{V}$ as the *trusted technology node*; likewise, the technology node of $\mathcal{P}$ is the *untrusted technology node*.

During operation, the prover (purportedly) executes Ψ, given a run-time input x; $\mathcal{P}$ returns the (purported) output y to $\mathcal{V}$. For each execution, $\mathcal{P}$ and $\mathcal{V}$ engage in a protocol. If y is the correct output and $\mathcal{P}$ follows the protocol, $\mathcal{V}$ must accept; otherwise, $\mathcal{V}$ must reject with high probability.

$\mathcal{P}$ can deviate arbitrarily from the protocol. However, it is assumed to be a polynomial-time adversary and is thus subject to standard cryptographic hardness assumptions (it cannot break encryption, etc.). This models multiple cases: $\mathcal{P}$ could have been designed maliciously, manufactured with an arbitrarily modified design [105], replaced with a counterfeit [10, 65] en route to the principal's trust domain, and so on.

Performance and costs. The cost of $\mathcal{V}$ and $\mathcal{P}$ together must be less than the **native baseline** (§1): a chip or CPU, in the same technology node as $\mathcal{V}$, that executes Ψ directly. By "cost", we mean general hardware considerations: energy consumption, chip area and packaging, throughput, non-recurring manufacturing costs, etc. Also, $\mathcal{V}$ and $\mathcal{P}$ must be physically realizable in the first place. Note that costs and physical realizability depend on the computation Ψ, the design of $\mathcal{V}$, the design of $\mathcal{P}$, and the technology nodes of $\mathcal{V}$ and $\mathcal{P}$.

Integration assumptions. There are requirements that surround integration; our work does not specifically address them, so we regard them as assumptions. First, $\mathcal{P}$'s influence on $\mathcal{V}$ should be limited to the defined communication interface, otherwise $\mathcal{P}$ could undermine or disable $\mathcal{V}$; similarly, if $\mathcal{V}$ has private state, then $\mathcal{P}$ should not have access to that state. Second, $\mathcal{P}$ must be isolated from all components besides $\mathcal{V}$. This is because $\mathcal{P}$'s outputs are untrusted; also, depending on the protocol and architecture, exfiltrating past messages could affect soundness for other verifier-prover pairs (§4). These two requirements might call for separate power supplies, shielding $\mathcal{V}$ from electromagnetic and side-channel inference attacks [121], etc. However, we acknowledge that eliminating side and covert channels is its own topic. Third, $\mathcal{V}$ may need to include a fail-safe, such as a kill switch, in case $\mathcal{P}$ returns wrong answers or refuses to engage.

2.2 Interactive proofs for verifiable computation

In designing a $\mathcal{V}$ and $\mathcal{P}$ that respond to the problem statement, our starting point is a protocol that we call *OptimizedCMT*. Using an observation of Thaler [107] and additional simplifications and optimizations, this protocol—which we are not claiming as a contribution of this paper—optimizes Allspice's CMT-slim protocol [112, §3], which refines CMT [55, §3; 108], which refines GKR [63, §3]; these are all interactive proofs [22, 63, 64, 80, 102]. We start with OptimizedCMT because, as noted in Section 1, it has good parallelism and data locality, qualities that ought to lead to physical realizability and efficiency in energy, area, and throughput. In the rest of this section, we describe OptimizedCMT; this description owes some textual debts to Allspice [112].

A verifier $\mathcal{V}$ and a prover $\mathcal{P}$ agree, offline, on a computation Ψ, which is expressed as an *arithmetic circuit (AC)* $\mathscr{C}$. In the arithmetic circuit formalism, a computation is represented as a set of abstract "gates" corresponding to field operations (add and multiply) in a given finite field, $\mathbb{F} = \mathbb{F}_p$ (the integers mod a prime p); a gate's two input "wires" and its output "wire" represent values in $\mathbb{F}$. OptimizedCMT requires that the AC be *layered*: the inputs connect only to first level gates, the outputs of those gates connect only to second-level gates, and so on. Denote the number of layers d and the number of gates in a layer G; we assume, for simplicity, that all layers, except for the input and output, have the same number of gates.

The aim of the protocol is for $\mathscr{P}$ to prove to $\mathscr{V}$ that, for a given input x, a given purported output vector y is truly $\mathscr{C}(x)$. The high-level idea is, essentially, cross examination: $\mathscr{V}$ draws on randomness to ask $\mathscr{P}$ unpredictable questions about the state of each layer. The answers must be consistent with each other and with y and x, or else $\mathscr{V}$ rejects. The protocol achieves the following; the probabilities are over $\mathscr{V}$'s random choices:

- **Completeness.** If $y = \mathscr{C}(x)$ and if $\mathscr{P}$ follows the protocol, then $\Pr\{\mathscr{V} \text{ accepts}\} = 1$.
- **Soundness.** If $y \neq \mathscr{C}(x)$, then $\Pr\{\mathscr{V} \text{ accepts}\} < \varepsilon$, where $\varepsilon = (\lceil \log |y| \rceil + 5d \log G)/|\mathbb{F}|$ and $|\mathbb{F}|$ is typically a large prime. This is an unconditional guarantee: it holds regardless of the prover's computational resources and strategy. This follows from the analysis in GKR [63, §2.5,§3.3], applied to Thaler's observations [107] (see also [112, §A.2]).
- **Efficient verifier.** Whereas the cost of executing $\mathscr{C}$ directly would be $O(d \cdot G)$, the verifier's online work is of lower complexity: $O(d \cdot \log G + |x| + |y|)$. This assumes that $\mathscr{V}$ has access to *auxiliary inputs* that have been precomputed offline; auxiliary inputs are independent of the input to $\mathscr{C}$. For each run of the protocol, an auxiliary input is of size $O(d \cdot \log G)$; the time to generate one is $O(d \cdot G)$.
- **Efficient prover.** The prover's work is $O(d \cdot G \cdot \log G)$.

Applicability. Consider a computation Ψ expressed in any model: binary executable on CPU, Boolean circuit, ASIC design, etc. Excluding precomputation, OptimizedCMT saves work for $\mathscr{V}$ versus executing Ψ in its native model, provided (1) Ψ can be represented as a deterministic AC, and (2) the AC is sufficiently wide relative to its depth ($G \gg d$). Requirement (2) can be relaxed by executing many copies of a narrow arithmetic circuit in parallel (§8.2).

Requirement (1) further interferes with applicability. One reason is that although any computation Ψ can in theory be represented as a deterministic AC, if Ψ has comparisons, bitwise operations, or indirect memory addressing, the resulting AC is very large. Prior work [31, 33, 88, 99, 101, 114] handles these program constructs compactly by exploiting *non-deterministic* ACs, and Allspice [112, §4.1] applies those techniques to the ACs that OptimizedCMT works over. Provided Ψ invokes a relatively small number—sub-linear in the running time—of these constructs, $\mathscr{V}$ can potentially save work [112, §4.3].

Also, ACs work over a field, usually $\mathbb{F}_p$; if Ψ were expressed over a different domain (for example, fixed-width integers, as on a CPU), the AC would inflate (relative to Ψ), ruling out work-savings for reasonably-sized computations. This issue plagues all built systems for verifiable computation [24, 25, 29, 31–33, 41, 51, 55, 56, 58, 59, 61, 76, 88, 98–101, 106, 108, 112, 114] (§9). This paper sidesteps it by assuming that Ψ is expressed over $\mathbb{F}_p$. That is, to be conservative, we restrict focus to computations that are naturally expressed as ACs.

Protocol details. Within a layer of the arithmetic circuit, gates are numbered between 1 and G and have a *label* corresponding to the binary representation of their number, viewed as an element of $\{0,1\}^b$, where $b = \lceil \log G \rceil$.

The AC's layers are numbered in reverse order of execution, so its inputs (x) are inputs to the gates at layer $d-1$, and its outputs (y) are viewed as being at layer 0. For each $i = 0, \ldots, d$, the *evaluator function* $V_i \colon \{0,1\}^b \to \mathbb{F}$ maps a gate's label to the correct output of that gate; these functions are particular to execution on a given input x. Notice that $V_d(j)$ returns the j^{th} input element, while $V_0(j)$ returns the j^{th} output element.

Observe that $\mathscr{C}(x) = y$, meaning that y is the correct output, if and only if $V_0(j) = y_j$ for all output gates j. However, $\mathscr{V}$ cannot check directly whether this condition holds: evaluating $V_0(\cdot)$ would require re-executing the AC, which is ruled out by the problem statement. Instead, the protocol allows $\mathscr{V}$ to efficiently reduce a condition on $V_0(\cdot)$ to a condition on $V_1(\cdot)$. That condition also cannot be checked—it would require executing most of the AC—but the process can be iterated until it produces a condition that $\mathscr{V}$ can check directly.

This high-level idea motivates us to express $V_{i-1}(\cdot)$ in terms of $V_i(\cdot)$. To this end, define the *wiring predicate* $\text{add}_i \colon \{0,1\}^{3b} \to \mathbb{F}$, where $\text{add}_i(g, z_0, z_1)$ returns 1 if g is an add gate at layer $i-1$ whose inputs are z_0, z_1 at layer i, and 0 otherwise; mult_i is defined analogously for multiplication gates. Now, $V_{i-1}(g) = \sum_{z_0,z_1 \in \{0,1\}^b} \text{add}_i(g, z_0, z_1) \cdot (V_i(z_0) + V_i(z_1)) + \text{mult}_i(g, z_0, z_1) \cdot V_i(z_0) \cdot V_i(z_1)$.

An important concept is *extensions*. An extension of a function f is a function $\tilde{f}$ that: works over a domain that encloses the domain of f, is a polynomial, and matches f everywhere that f is defined. In our context, given a function $g \colon \{0,1\}^m \to \mathbb{F}$, the *multilinear extension* (it is unique) $\tilde{g} \colon \mathbb{F}^m \to \mathbb{F}$ is a polynomial that agrees with g on its domain and that has degree at most one in each of its m variables. Throughout this paper, we notate multilinear extensions with tildes. Thaler [107], building on GKR [63], shows:

$$\tilde{V}_{i-1}(q) = \sum_{z_0,z_1 \in \{0,1\}^b} P_q(z_0, z_1), \text{ where} \tag{1}$$

$$P_q(z_0, z_1) = \tilde{\text{add}}_i(q, z_0, z_1) \cdot \left(\tilde{V}_i(z_0) + \tilde{V}_i(z_1)\right) + \tilde{\text{mult}}_i(q, z_0, z_1) \cdot \tilde{V}_i(z_0) \cdot \tilde{V}_i(z_1),$$

with signatures $\tilde{V}_i, \tilde{V}_{i-1} \colon \mathbb{F}^b \to \mathbb{F}$ and $\tilde{\text{add}}_i, \tilde{\text{mult}}_i \colon \mathbb{F}^{3b} \to \mathbb{F}$. Also, $P_q \colon \mathbb{F}^{2b} \to \mathbb{F}$. (GKR's expression for $\tilde{V}_{i-1}(\cdot)$ sums a $3b$-variate polynomial over G^3 terms. Thaler's $2b$-variate polynomial, P_q, summed over G^2 terms, reduces the prover's runtime, rounds, and communication by about 33%.)

At this point, the form of $\tilde{V}_{i-1}(\cdot)$ calls for a *sumcheck protocol* [80]: an interactive protocol in which a prover establishes for a verifier a claim about the sum, over a hypercube, of a given polynomial's evaluations. Here, the polynomial is P_q.

For completeness, we give the entire protocol between $\mathscr{P}$ and $\mathscr{V}$, in Figure 2. Many of the details are justified elsewhere [63, §3][55, §A.1–A.2][112, §2.2,§A][107]. Here, our

```
1:  function PROVE(ArithCircuit c, input x)
2:    q_0 ← ReceiveFromVerifier()    // see line 56
3:    d ← c.depth
4:
5:    // each circuit layer induces one sumcheck invocation
6:    for i = 1,...,d do
7:      w_0, w_1 ← SUMCHECKP(c, i, q_{i-1})
8:      τ_i ← ReceiveFromVerifier()    // see line 71
9:      q_i ← (w_1 − w_0) · τ_i + w_0
10:
11: function SUMCHECKP(ArithCircuit c, layer i, q_{i-1})
12:   for j = 1,...,2b do
13:
14:     // compute F_j(0), F_j(1), F_j(2)
15:     parallel for all gates g at layer i − 1 do
16:       for k = 0,1,2 do
17:         // below, s ∈ {0,1}^{3b}. s is a gate triple in binary.
18:         s ← (g, g_L, g_R) // g_L, g_R are labels of g's layer-i inputs
19:
20:         u_k ← (q_{i-1}[1],...,q_{i-1}[b], r[1],...,r[j−1], k)
21:         // notation: χ : F → F. χ_1(t) = t, χ_0(t) = 1−t
22:         termP ← ∏_{ℓ=1}^{b+j} χ_{s[ℓ]}(u_k[ℓ])
23:
24:         if j ≤ b then
25:           termL ← Ṽ_i(r[1],...,r[j−1], k, g_L[j+1],...,g_L[b])
26:           termR ← V_i(g_R) // V_i = Ṽ_i on gate labels
27:         else    // b < j ≤ 2b
28:           termL ← Ṽ_i(r[1],...,r[b])
29:           termR ← Ṽ_i(r[b+1],...,r[j−1], k,
30:                       g_R[j−b+1],...,g_R[b])
31:
32:         if g is an add gate then
33:           F[g][k] ← termP · (termL + termR)
34:         else if g is a mult gate then
35:           F[g][k] ← termP · termL · termR
36:
37:     for k = 0,1,2 do
38:       F_j[k] ← Σ_{g=1}^{G} F_j[g][k]
39:
40:     SendToVerifier(F_j[0], F_j[1], F_j[2])    // see line 82
41:     r[j] ← ReceiveFromVerifier()    // see line 87
42:
43:   // notation
44:   w_0 ← (r[1],...,r[b])
45:   w_1 ← (r[b+1],...,r[2b])
46:
47:   SendToVerifier(Ṽ_i(w_0), Ṽ_i(w_1))    // see line 99
48:
49:   for t = {2,...,b},    w_t ← (w_1 − w_0) · t + w_0
50:   SendToVerifier(Ṽ_i(w_2),...,Ṽ_i(w_b))    // see line 67
51:
52:   return (w_0, w_1)

53: function VERIFY(ArithCircuit c, input x, output y)
54:   q_0 ←R F^b
55:   a_0 ← Ṽ_y(q_0)    // Ṽ_y(·) is the multilinear ext. of the output y
56:   SendToProver(q_0)    // see line 2
57:   d ← c.depth
58:
59:   for i = 1,...,d do
60:     // reduce a_{i-1} ?= Ṽ_{i-1}(q_{i-1}) to h_0 ?= Ṽ_i(w_0), h_1 ?= Ṽ_i(w_1) ...
61:     (h_0, h_1, w_0, w_1) ← SUMCHECKV(i, q_{i-1}, a_{i-1})
62:
63:     // ... and reduce h_0 ?= Ṽ_i(w_0), h_1 ?= Ṽ_i(w_1) to a_i ?= Ṽ_i(q_i).
64:     // we will interpolate H(t) = Ṽ_i((w_1 − w_0)t + w_0), so we want
65:     // H(0),...,H(b). But h_0, h_1 should be H(0), H(1), so we now
66:     // need H(2),...,H(b)
67:     h_2,...,h_b ← ReceiveFromProver()    // see line 50
68:     τ_i ←R F
69:     q_i ← (w_1 − w_0) · τ_i + w_0
70:     a_i ← H*(τ_i) // H* is poly. interpolation of h_0, h_1, h_2,..., h_b
71:     SendToProver(τ_i)    // see line 8
72:
73:   if a_d = Ṽ_d(q_d)  then    // Ṽ_d(·) is multilin. ext. of the input x
74:     return accept
75:   return reject
76:
77: function SUMCHECKV(layer i, q_{i-1}, a_{i-1})
78:   r ←R F^{2b}
79:   e ← a_{i-1}
80:   for j = 1,2,...,2b do
81:
82:     F_j[0], F_j[1], F_j[2] ← ReceiveFromProver()    // see line 40
83:
84:     if F_j[0] + F_j[1] ≠ e then
85:       return reject
86:
87:     SendToProver(r[j])    // see line 41
88:
89:     reconstruct F_j(·) from F_j[0], F_j[1], F_j[2]
90:     // F_j(·) is degree-2, so three points are enough.
91:
92:     e ← F_j(r[j])
93:
94:   // notation
95:   w_0 ← (r[1],...,r[b])
96:   w_1 ← (r[b+1],...,r[2b])
97:
98:   // P is supposed to set h_0 = Ṽ_i(w_0) and h_1 = Ṽ_i(w_1)
99:   h_0, h_1 ← ReceiveFromProver()    // see line 47
100:  a' ← add̃_i(q_{i-1}, w_0, w_1)(h_0 + h_1) + m̃ult_i(q_{i-1}, w_0, w_1) h_0 · h_1
101:  if a' ≠ e then
102:    return reject
103:  return (h_0, h_1, w_0, w_1)
```

FIGURE 2—Pseudocode for $\mathscr{P}$ and $\mathscr{V}$ in OptimizedCMT [55, 63, 107, 112], expressed in terms of the parallelism used by TRMP [108]. Some of the notation and framing is borrowed from Allspice [112]. The protocol proceeds in layers. At each layer i, the protocol reduces the claim that $a_{i-1} = \tilde{V}_{i-1}(q_{i-1})$ to a claim that $a_i = \tilde{V}_i(q_i)$; note that equation (1) in the text expresses $\tilde{V}_{i-1}(q_{i-1})$ as a sum over a hypercube. The SumCheck sub-protocol guarantees to $\mathscr{V}$ that, with high probability, this sum equals a_{i-1} if and only if $h_0{=}\tilde{V}_i(w_0)$ and $h_1{=}\tilde{V}_i(w_1)$; an additional reduction connects these two conditions to the claim that $a_i = \tilde{V}_i(q_i)$. See [63, §3][55, §A.1–A.2][112, §2.2,§A][107] for explanation of all details.

aim is to communicate the structure of the protocol and the work that $\mathscr{P}$ performs. There is one *invocation* of the sum-check protocol for each layer of the arithmetic circuit. Within an invocation, there are $2b$ *rounds* (logarithmic in layer width).

In each round j, $\mathscr{P}$ describes to $\mathscr{V}$ a univariate polynomial:

$$F_j(t^*) = \sum_{t_{j+1},\ldots,t_{2b} \in \{0,1\}^{2b-j}} P_{q_{i-1}}(r_1,\ldots,r_{j-1},t^*,t_{j+1},\ldots,t_{2b}),$$

where r_j is a random challenge sent by $\mathscr{V}$ at the end of round j, and q_{i-1} is a function of random challenges in prior invocations. $F_j(\cdot)$ is degree 2, so $\mathscr{P}$ describes it by sending three evaluations—$F_j(0), F_j(1), F_j(2)$—which $\mathscr{V}$ interpolates to obtain the coefficients of $F_j(\cdot)$. How does $\mathscr{P}$ compute these evaluations? Naively, this seems to require $\Omega(G^3 \log G)$ operations. However, CMT [55, §A.2] observes that $F_j(k)$ can be written as a sum with one term per gate, in the form:

$$F_j(k) = \sum_{g \in S_{\text{add},i}} \text{termP}_{j,g,k} \cdot (\text{termL}_{j,g,k} + \text{termR}_{j,g,k}) + \sum_{g \in S_{\text{mult},i}} \text{termP}_{j,g,k} \cdot \text{termL}_{j,g,k} \cdot \text{termR}_{j,g,k},$$

where $S_{\text{add},i}$ (resp., $S_{\text{mult},i}$) has one element for each add (resp., multiplication) gate g at layer $i-1$. The definitions of termP, termL, and termR are given in Figure 2; these terms depend on j, on g, on k, and on prior challenges.

Recall that $\mathscr{V}$ receives an auxiliary input for each run of the protocol. An auxiliary input has two components. The first one is $O(d \cdot \log G)$ field elements: the evaluations of $\tilde{\text{add}}_i(q_{i-1}, w_0, w_1)$ and $\tilde{\text{mult}}_i(q_{i-1}, w_0, w_1)$, for $i = \{1,\ldots,d\}$ (line 100, Figure 2), together with τ_i, w_0, and w_1. As shown elsewhere [112, §A.3], $\tilde{\text{add}}_i(t_1,\ldots,t_{3b}) = \sum_{s \in \{0,1\}^{3b}:\ \text{add}_i(s)=1} \prod_{\ell=1}^{3b} \chi_{s[\ell]}(t_\ell)$, and analogously for $\tilde{\text{mult}}_i$. Computing these quantities naively would require $3 \cdot G \cdot \log G$ operations; an optimization of Allspice [112, §5.1] lowers this to between $12 \cdot G$ and $15 \cdot G$ operations and thus $O(d \cdot G)$ for all layers. The second component comprises Lagrange basis coefficients [16], used to lower the "online" cost of interpolating H^* (line 70, Figure 2) from $O(\log^2 G)$ to $O(\log G)$. The size of this component is $O(d \cdot \log G)$ field elements; the time to compute it is $O(d \cdot \log^2 G)$. We discuss how one can amortize these precomputations in Section 4.[3]

2.3 Hardware considerations and metrics

The high-level performance aim was described earlier (§2.1); below, we delve into the specific evaluation criteria for Zebra.

With hardware, choosing metrics is a delicate task. One reason is that some metrics can be "fooled". For example, per-chip manufacturing costs are commonly captured by *area* (the size of an ASIC in square millimeters). Yet area alone is an incomplete metric: a design might iteratively re-use modules to lower area, but that would also lower throughput. Another issue is that costs are multifaceted: one must also account for operating costs and hence *energy consumption* (joules/operation). Finally, physics imposes constraints, described shortly.

We will use two metrics, under two constraints. The first metric is energy per protocol run, denoted E, which captures both the number of basic operations (arithmetic, communication, etc.) in each run and the efficiency of each operation's implementation. The second metric is the ratio of area and throughput, denoted A_s/T. The term A_s is a weighted sum of area consumed in the trusted and untrusted technology nodes; the untrusted area is divided by s. We leave s as a parameter because (a) s will vary with the principal, and (b) comparing costs across foundries and technology nodes is a thorny topic, and arguably a research question [43, 82]. For both metrics, lower is better.

Turning to constraints, the first is a ceiling on area for any chip: $\mathscr{P}$, $\mathscr{V}$, and the baseline. This reflects manufacturability: large chips have low manufacturing yield owing to defects. Constraining area affects A_s/T because it rules out, for example, designs that abuse some super-linear throughput improvement per unit area. The second constraint is a limit on power dissipation, because otherwise heat becomes an issue. This constrains the product of E and T (which is in watts), bounding parallelism by restricting the total number of basic operations that can be executed in a given amount of time.

In measuring Zebra's performance, we will try to capture all costs associated with executing OptimizedCMT. As an important example, we will include not only $\mathscr{V}$'s and $\mathscr{P}$'s computation but also their interaction (§4); this is crucial because the protocol entails significant communication overhead. We model costs in detail in Section 5.

We discuss non-recurring costs (engineering, creating photolithographic masks, etc.) in Section 9.

3 Design of prover and verifier in Zebra

This section details the design of Zebra's hardware prover; we also briefly describe Zebra's hardware verifier. The designs follow OptimizedCMT and thus inherit its completeness, soundness, and asymptotic efficiency (§2.2). The exception is that $\mathscr{P}$'s design has an additional $O(\log G)$ factor in running time (§3.2). To achieve physically realizable, high-throughput, area-efficient designs, Zebra exploits new and existing observations about OptimizedCMT. Zebra's design ethos is:

- *Extract parallelism.* This does not reduce the total work that must be done, but it does lead to speedups. Specifically, for a given area, more parallelism yields better throughput.
- *Exploit locality.* This means avoiding unnecessary communication among modules: excessive communication constrains physical realizability. Locality also refers to avoiding dependencies among modules; in hardware, dependencies

[3] CMT [55, 106, 108] avoids amortization by restricting focus to *regular* arithmetic circuits, meaning that $\tilde{\text{add}}_i$ and $\tilde{\text{mult}}_i$ can be computed in $O(\text{polylog}(G))$ time. In that setup, $\mathscr{V}$ incurs this cost "online", at which point there is no reason to precompute the Lagrange coefficients either.

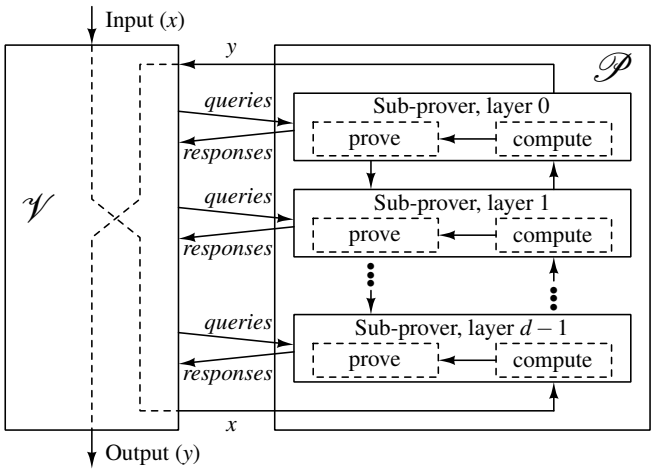


FIGURE 3—Architecture of prover ($\mathscr{P}$) in Zebra. Each logical sub-prover handles all of the work for a layer in a layered arithmetic circuit $\mathscr{C}$ of width G and depth d.

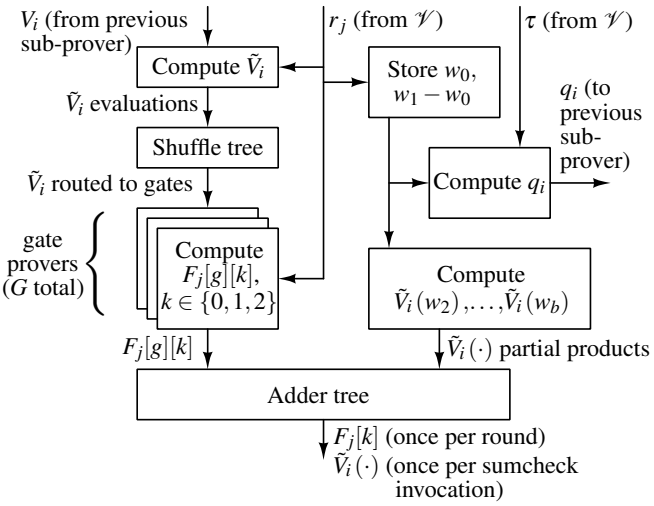


FIGURE 4—Design of a sub-prover.

translate to timing relationships,[4] timing relationships create serialization points, and serialization harms throughput.

- *Reuse work.* This means both reusing computation results (saving energy), and reusing modules in different parts of the computation (saving area). This reuse must be carefully engineered to avoid interfering with the previous two goals.

3.1 Overview

Figure 3 depicts the top-level design of the prover. The prover comprises logically separate *sub-provers*, which are multiplexed over physical sub-prover modules. Each logical sub-prover is responsible for executing a layer of $\mathscr{C}$ and for the proof work that corresponds to that layer (lines 7–9 in Figure 2). The execution runs forward; the proving step happens in the opposite order. As a consequence, each sub-prover must buffer the results of executing its layer, until those results are used in the corresponding proof step. Zebra is agnostic about the number of physical sub-prover modules; the choice is an area-throughput trade-off (§3.2).

The design of a sub-prover is depicted in Figure 4. A sub-prover proceeds through sumcheck rounds sequentially, and reuses the relevant functional blocks over every round of the protocol (which contributes to area efficiency). Within a round, *gate provers* work in parallel, roughly as depicted in Figure 2 (line 15), though Zebra extracts additional parallelism (§3.2). From one round to the next, intermediate results are preserved locally at each functional unit (§3.3). At lower levels of the design, the sub-prover has a latency-insensitive control structure (§3.3), and the sub-prover avoids work by reusing computation results (§3.4).

[4] *Synthesis* is the process of translating a digital circuit design into concrete logic gates. This translation must satisfy relationships, known as *timing constraints*, between signals. Constraints are explicit (e.g., the designer specifies the frequency of the master clock) and implicit (e.g., one flip-flop's input relies on a combinational function of another flip-flop's output).

The design of the verifier is similar to that of the prover; however, the verifier's work is more serial, so its design aims to reuse modules, and thus save area, without introducing bottlenecks (§3.5).

The rest of this section delves into details, highlighting the innovations; our description is roughly organized around the ethos presented earlier.

3.2 Pipelining and parallelism

The pseudocode for $\mathscr{P}$ (Figure 2) is expressed in terms of the parallelism that has been observed before [108]. Below, we describe further parallelism extracted by Zebra's design.

Exploiting layered arithmetic circuits. The required layering in the arithmetic circuit (§2.2) creates a textbook trade-off between area and throughput. At one extreme, Zebra can conserve area, by having a single physical sub-prover, which is iteratively reused for each layer of the AC. The throughput is given by the time to execute and prove each layer of $\mathscr{C}$.

At the other extreme, Zebra can spend area, dedicate a physical sub-prover to each layer of the arithmetic circuit, and arrange them in a classical pipeline. Specifically, the sub-prover for layer i handles successive *runs*, in each "epoch" always performing the proving work of layer i, and handing its results to the sub-prover for layer $i+1$. The parallelism that is exploited here is that of multiple runs; the acceleration in throughput, compared to using a single physical sub-prover, is d. That is, Zebra's prover can produce proofs at a rate determined only by the time taken to prove a single layer.

Zebra also handles intermediate points, such as two logical sub-provers per physical prover. In fact, Zebra can have more runs in flight than there are physical sub-provers. Specifically, instead of handling several layers per run and then moving to the next run, a physical sub-prover can iterate over runs while keeping the layer fixed, then handle another layer and again iterate over runs.

The source of Zebra's flexibility is that in both execution and

proving, there are narrow, predictable dependencies between layers (which itself stems from OptimizedCMT's requirement to use a layered AC). As previously mentioned, a sub-prover must buffer the results of executing a layer until the sub-prover has executed the corresponding proof step. $\mathscr{P}$'s total buffer size is the number of in-flight runs times the size of $\mathscr{C}$.

Gate-level parallelism. Within a sub-prover, and within a sumcheck round, there is parallel proving work not only for each gate (as in prior work [108]) but also for each (gate, k) pair, for $k = \{0,1,2\}$. That is, in Zebra, the loops in lines 15 and 16 (Figure 2) are combined into a "parallel for all (g,k)". This is feasible because, loosely speaking, sharing state read-only among modules requires only creating wires, whereas in a traditional memory architecture, accesses are serialized.

The computation of termP (line 22) is an example. To explain it, we first note that each gate prover stores state, which we notate P_g. Now, for a gate prover g, let $s = (g, g_L, g_R)$, where g_L, g_R are the labels of g's inputs in $\mathscr{C}$. (We have used g to refer both to a gate and the gate prover for that gate; throughout our description, each gate prover will be numbered and indexed identically to its corresponding gate in $\mathscr{C}$.) P_g is initialized to $\prod_{\ell=1}^{b} \chi_{s[\ell]}(q[1],\ldots,q[b])$; at the end of a sumcheck round j, each gate prover updates P_g by multiplying it with $\chi_{s[b+j]}(r[j])$. At the beginning of a round j, P_g is shared among the $(g,0)$, $(g,1)$, and $(g,2)$ functional blocks, permitting simultaneous computation of termP (by multiplying P_g with $\chi_{s[b+j]}(k)$).

Removing the $\tilde{V}_i(w_2),\ldots,\tilde{V}_i(w_b)$ bottleneck. At the end of a sumcheck invocation, the prover has an apparent bottleneck: computing $\tilde{V}_i(w_2),\ldots,\tilde{V}_i(w_b)$ (line 50 in Figure 2). However, Zebra observes that these quantities can be computed in parallel with the rest of the invocation, using incremental computation and local state. To explain, we begin with the form of $\tilde{V}_i$:[5]

$$\tilde{V}_i(q) = \sum_{g=1}^{G} V_i(g) \prod_{\ell=1}^{b} \chi_{g[\ell]}(q[\ell]), \tag{2}$$

where $q \in \mathbb{F}^b$, $g[\ell]$ is the ℓ^{th} bit of the binary expansion of gate g, and $q[\ell]$ is the ℓ^{th} component of q.

Prior work [108] performs the evaluations of $\tilde{V}_i(w_2),\ldots,\tilde{V}_i(w_b)$ using $O(G \cdot \log G)$ operations, at the end of a sumcheck invocation. Indeed, at first glance, the required evaluations seem as though they can be done only after line 42 (Figure 2) because for $t > 2$, w_t depends on w_0, w_1 (via $w_t \leftarrow (w_1 - w_0) \cdot t + w_0$), which are not fully available until the end of the outer loop.

However, we observe that all of w_0 is available after round b, and w_1 is revealed over the remaining rounds. Zebra's prover exploits this observation to gain parallelism. The tradeoff is more work overall: $(b-1) \cdot G \cdot b$ products, or $O(G \cdot \log^2 G)$.

[5] One knows that $\tilde{V}_i$, the multilinear extension of V_i, has this form because this expression is a multilinear polynomial that agrees with V_i everywhere that V_i is defined, and because multilinear extensions are unique.

In detail, after round $b+1$, $\mathscr{P}$ has $w_1[1]$ (because this is just $r[b+1]$, which is revealed in line 41, Figure 2). $\mathscr{P}$ also has $w_0[1]$ (because w_0 has been fully revealed). Thus, $\mathscr{P}$ has, or can compute, $w_0[1],\ldots,w_b[1]$ (by definition of w_t). Given these, $\mathscr{P}$ can compute $V_i(g) \cdot \chi_{g[1]}(w_t[1])$, for $g \in \{1,\ldots,G\}$ and $t \in \{2,\ldots,b\}$.

Similarly, after round $b+2$, $r[b+2]$ is revealed; thus, using the $(b-1) \cdot G$ products from the prior round, $\mathscr{P}$ can perform another $(b-1) \cdot G$ products to compute $V_i(g) \cdot \prod_{\ell=1}^{2} \chi_{g[\ell]}(w_t[\ell])$, for $g \in \{1,\ldots,G\}$, $t \in \{2,\ldots,b\}$. This process continues, until $\mathscr{P}$ is storing $V_i(g) \cdot \prod_{\ell=1}^{b} \chi_{g[\ell]}(w_t[\ell])$, for all g and all t, at which point, summing over the g is enough to compute $\tilde{V}_i(w_t)$, by Equation (2).

Zebra's design exploits this observation with G circular shift registers. For each update (that is, each round j, $j > b$), and in parallel for each $g \in \{1,\ldots,G\}$, Zebra's sub-prover reads a value from the head of the g^{th} circular shift register, multiplies it with a new value, replaces the previous head value with the product, and then circularly shifts, thereby yielding the next value to be operated on. For each g, this happens $b-1$ times per round, with the result that at round j, $j > b$, $V_i(g) \cdot \prod_{\ell=1}^{j-b-1} \chi_{g[\ell]}(w_t[\ell])$ is multiplied by $\chi_{g[j-b]}(w_t[j-b])$, for $g \in \{1,\ldots,G\}$, $t \in \{2,\ldots,b\}$. At the end, for each t, the sum over all g uses a tree of adders.

3.3 Extracting and exploiting locality

Locality of data. Zebra's sub-prover must avoid RAM, because it would cause serialization bottlenecks. Beyond that, Zebra must avoid globally consumed state, because it would add global communication, which has the disadvantages noted earlier. These points imply that, where possible, Zebra should maintain state "close" in time and space to where it is consumed; also, the paths from producer-to-state, and from state-to-consumers should follow static wiring patterns. We have already seen two examples of this, in Section 3.2: (a) the P_g values, which are stored in accumulators within the functional blocks of the gate provers that need them, and (b) the circular shift registers. Below, we present a further example, as a somewhat extreme illustration.

A key task for $\mathscr{P}$ in each round of the sumcheck protocol is to compute termL and termR (lines 25 through 30, Figure 2), by evaluating $\tilde{V}_i(\cdot)$ at various points. Prior work [55, 108, 112] performs this task efficiently, by incrementally computing a *basis*: at the beginning of each round j, $\mathscr{P}$ holds a lookup table that maps each of the 2^{b-j+1} hypercube vertexes $T_j = (t_j,\ldots,t_b)$ to $\tilde{V}(r[1],\ldots,r[j-1],T_j)$, where $t_1,\ldots,t_b$ denote bits. (We are assuming for simplicity that $j \le b$; for $j > b$, the picture is similar.) Then, for each (g,k), the required termL can be read out of the table, by looking up the entries indexed by $(k, g_L[j+1],\ldots,g_L[b])$ for $k = 0,1$.[6] This requires updating, and shrinking, the lookup table at the end of each round, a step that can be performed efficiently because for all

[6] These values (multiplying one by 2 and the other by -1, and summing) also yield termL for $k = 2$, which is a consequence of Equation (2).

$T_{j+1} = (t_{j+1}, \ldots, t_b) \in \mathbb{F}_2^{b-j}$, Equation (2) implies:

$$\tilde{V}_i(r[1], \ldots, r[j], T_{j+1}) = (1 - r[j]) \cdot \tilde{V}_i(r[1], \ldots, r[j-1], 0, T_{j+1}) + r[j] \cdot \tilde{V}_i(r[1], \ldots, r[j-1], 1, T_{j+1}).$$

Zebra must implement equivalent logic, meaning on-demand "lookup" and incremental update, but without random-access state. To do so, Zebra maintains $2^b \approx G$ registers; each is initialized as in prior work, with $\tilde{V}_i(t_1, \ldots, t_b) = V_i(t_1, \ldots, t_b)$, for all $(t_1, \ldots, t_b) \in \mathbb{F}_2^b$. The update step relies on a static wiring pattern: roughly speaking, each entry T is updated based on $2T$ and $2T+1$. Then, for the analog of the "lookup," Zebra delivers the basis values to the gate provers that need them. This step uses what we call a *shuffle tree*: a tree of multiplexers in which the basis values are inputs to the tree at multiple locations, and the outputs are taken from a different level of the tree at each round j. The effect is that, even though the basis keeps changing—by the end, it is only two elements—the required elements are sent to all of the gate provers.

Locality of control. Zebra's prover must orchestrate the work of execution and proving. The naive approach would be a top-level state machine controlling every module. However, this approach would destroy locality (control wires would be sent throughout the design) and create inefficiencies (not all modules have the same timing, leading to idling and waste).

Instead, Zebra's prover has a *latency-insensitive design*. There is a top-level state machine, but it handles only natural serialization points, such as communication with the verifier. Otherwise, Zebra's modules are arranged in a hierarchical structure: at all levels of the design, parents send signals to children indicating that their inputs are valid, and children produce signals indicating valid outputs. These are the only timing relations, so control wires are local, and go only where needed. As an example, within a round of the sumcheck protocol, a sub-prover instructs all gate provers g to begin executing; when the outputs are ready, the sub-prover feeds them to an adder tree to produce the required sum (line 38, Figure 2). Even though gate provers complete their work at different times (owing to differences in, for example, mult and add), no additional control is required.

3.4 Reusing work

We have already described several ways in which a Zebra sub-prover reuses intermediate computations. But Zebra's sub-provers also reuse modules themselves, which saves area. For example, computing each of $F_j(0), F_j(1), F_j(2)$ uses an adder tree, as does computing $\tilde{V}_i(w_2), \ldots, \tilde{V}_i(w_b)$ (§3.2). But these quantities are never needed at the same time during the protocol. Thus, Zebra uses the *same* adder tree.

Something else to note is that nearly all of the sub-prover's work is field operations; indeed, all multiplications and additions in the *algorithm*, not just the AC $\mathscr{C}$, are field operations. This means that optimizing the circuits implementing these operations improves the performance of every module of $\mathscr{P}$.

3.5 Design of $\mathscr{V}$

In many respects, the design of Zebra's verifier is similar to the prover; for example, the approach to control is the same (§3.3). However, the verifier cannot adopt the prover's pipeline structure: for the verifier, different layers impose very different overheads. Specifically, the verifier's first and last layers are costly (lines 55, 73 in Figure 2), whereas the interior layers are lightweight (§2.2).

To address this issue, Zebra observes that the work of the first and last layers can happen in parallel; this work determines the duration of a pipeline stage's execution. Then, within that duration, interior layers can be computed sequentially. For example, two or more interior layers might be computed in the time it takes to compute just the input or output layer; the exact ratio depends on the length of the input and output, versus $\log G$. As noted earlier (§3.2), sequential work enables area savings; in this case the savings do not detract from throughput because the input and output layers are the bottleneck.

4 System architecture

This section articulates several challenges of system architecture and operation, and walks through Zebra's responses.

$\mathscr{V}$-$\mathscr{P}$ integration. The default integration approach—a printed circuit board—would limit bandwidth and impose high cost in energy for communication between $\mathscr{V}$ and $\mathscr{P}$. This would be problematic for Zebra because the protocol contains a lot of inter-chip communication (for $\mathscr{P}$, it is lines 8, 40, 41, 47, and 50; for $\mathscr{V}$, lines 67, 71, 82, 87, and 99). Zebra's response is to draw on *3D packaging* [75, 79], a technology that enables high bandwidth and low energy communication between chips (§7.1).

Is it reasonable to assume that the integrator (§2.1) has access to 3D packaging? We think yes: the precision requirements for 3D packaging are much less than for a transistor even in a very mature technology node. Moreover, this technology is commercially available and used in high-assurance contexts [13], albeit for a different purpose [70]. Finally, although 3D packaging is not yet in wide use, related packaging technologies are in high-volume production for commodity devices like inertial sensors and clock generators [90, 94].

Precomputation and amortization. All built systems for verifiable computation, except CMT applied to highly regular ACs, presume offline precomputation on behalf of the verifier that exceeds the work of simply executing the computation [24, 25, 29, 31–33, 41, 51, 55, 56, 58, 59, 61, 76, 88, 98–101, 106, 108, 112, 114]. They must therefore—if the goal is to save the verifier work—amortize the precomputation in one way or another. In Zebra, the precomputation is generating an auxiliary input (§2.2), which the operator supplies to $\mathscr{V}$ along with the input x. This arrangement raises two questions, which we answer below: (1) How does the operator get auxiliary inputs? (2) How does the work of generating them amortize?

We assume that the operator is given auxiliary inputs by

the integrator on a bulk storage medium (for example, a hard drive). When all auxiliary inputs on the medium are consumed, the integrator must securely refresh them (for example, by delivering another drive to the operator). The integrator must thus select the storage size according to the application. For example, at a throughput of 10^4 executions per second, and 10^4 bytes per auxiliary input, a 10TB drive suffices for 24 hours of continuous operation. Note that one store of precomputations can be shared among many instances of Zebra operating in close proximity (for example, in a data center).

As an optimization, an auxiliary input does not explicitly materialize τ_i, w_0, and w_1 for each layer (§2.2). Instead, $\mathcal{V}$ rederives them from a pseudorandom *seed* that is included in the auxiliary input. This reduces $\mathcal{V}$'s storage costs by roughly 2/3, to $d \cdot (\log G + 2) + 1$ field elements per auxiliary input.

To amortize the integrator's precomputation, Zebra presumes that the integrator reuses the precomputations over many $\mathcal{V}$; for example, over all $\mathcal{V}$ chips that are currently mated with a given $\mathcal{P}$ design.[7] Since precomputing an auxiliary input requires $O(d \cdot G)$ computation with good constants (§2.2), the overhead can be amortized to negligible across an entire deployment provided that there are, say, a few thousand operating verifiers. This assumes that the execution substrate for the precomputation is similar to $\mathcal{V}$.

Preserving soundness in this regime requires that each prover chip in an operator's system is isolated, a stipulation of the setup (§2.1). We further assume that operators run independently of one another, and that operators deploying multiple provers handle failures carefully. Specifically, if a verifier $\mathcal{V}$ rejects, then the auxiliary inputs that $\mathcal{V}$ has seen must now be viewed as disclosed, and not reused within the deployment. To explain, note that a prover $\mathcal{P}_\star$ can make its verifier, $\mathcal{V}_\star$, reject in a way that (say) correlates with the challenges that $\mathcal{V}_\star$ has sent. Meanwhile, if failed interactions influence the inputs received by other $\mathcal{V}$-$\mathcal{P}$ pairs, then $\mathcal{P}_\star$ has a (low-bandwidth) way to communicate information about challenges it has seen.[8] If $\mathcal{V}_\star$'s previously consumed auxiliary inputs continued to be used, then this channel would (slightly) reduce soundness.

Storing precomputations. The privacy and integrity of $\mathcal{V}$'s auxiliary inputs must be as trustworthy as $\mathcal{V}$ itself. Yet, storing auxiliary inputs in trustworthy media could impose enormous cost. Zebra's solution is to store auxiliary inputs in *untrusted* media (for example, a commercial hard drive) and to protect them with authenticated encryption. To this end, $\mathcal{V}$ and the integrator share an encryption key that $\mathcal{V}$ stores in a very small on-chip memory. When the integrator carries out its precomputation, it encrypts the result before storing it; upon receiving an auxiliary input, $\mathcal{V}$ authenticates and decrypts, rejecting the protocol run if tampering is detected. Note that Zebra does not need ORAM: the pattern of accesses is known.

$\mathcal{V}$'s interface to the operator. $\mathcal{V}$ needs a high-performance I/O interface to the operator to receive inputs (x) and auxiliary inputs, and send outputs (y), at throughput T. This requires bandwidth $T \cdot (|x| + |y| + d \cdot (\log G + 2) + 1)$, some tens or hundreds of megabits per second. Both area (for $\mathcal{V}$'s transceiver circuit) and energy (per bit sent) are concerns; Zebra's response is to use a high-performance optical backplane, which gives both good bandwidth per transceiver area and low, distance-insensitive energy cost [27, 87, 97]. While this technology is not yet widespread in today's datacenters, its use in high-performance computing applications [57, 71] indicates a trend toward more general deployment.

5 Cost analysis and accounting

This section presents an analytical model for the energy (E), area (A_s), and throughput (T) of Zebra when applied to ACs of depth d and width G. Figure 5 summarizes the model. It covers: (1) protocol execution (*compute*; §2.2, Fig. 2); (2) communication between $\mathcal{V}$ and $\mathcal{P}$ (*tx*; §2.2); (3) storage of auxiliary inputs for $\mathcal{V}$, including retrieval, I/O, and decryption for $\mathcal{V}$ (*store*; §4); (4) storage of intermediate pipeline values for $\mathcal{P}$ (*store*; §3.2); (5) pseudorandom number generation for $\mathcal{V}$ (*PRNG*; §4); and (6) communication between $\mathcal{V}$ and the operator ($\mathcal{V}$ *I/O*; §4). In Section 7.1, we derive estimates for the model's parameters using synthesis and simulation.

The following analogies hold: E captures the number of operations done by each of components (1)–(6) in a single execution of OptimizedCMT; A_s roughly captures the parallelism with which Zebra executes, in that it charges for the number of hardware modules and how they are allocated; and T captures the critical path of execution.

Energy and area costs. For both $\mathcal{P}$ and $\mathcal{V}$, all computations in a protocol run are expressed in terms of field arithmetic primitives (§3.4). As a result, field operations dominate energy and area costs for $\mathcal{P}$ and $\mathcal{V}$ (§7.2, §8). Many of Zebra's design decisions and optimizations (§3) show up in the constant factors in the energy and area "compute" rows.

Area and throughput trade-offs. Zebra's throughput is determined by the longest pipeline delay in either $\mathcal{V}$ or $\mathcal{P}$; that is, the pipeline delay of the faster component can be increased without hurting throughput. Zebra can trade increased pipeline delay for reduced area by removing functional units (e.g., sub-provers) in either $\mathcal{P}$ or $\mathcal{V}$ (§3.2, §3.5). Such trade-offs are typical in hardware design [84]; in Zebra, they are used to optimize A_s/T by balancing $\mathcal{V}$'s and $\mathcal{P}$'s delays (§7.4).

The model captures these trade-offs with particular parameters. For $\mathcal{P}$, these parameters are $n_{\mathcal{P},\text{sc}}$ and $n_{\mathcal{P},\text{pl}}$: the number of physical sub-prover modules and the number of in-flight runs (§3.2). For $\mathcal{V}$, these parameters are $n_{\mathcal{V},\text{sc}}$ and $n_{\mathcal{V},\text{io}}$: roughly, the area apportioned to sumcheck work and comput-

[7] Allspice, for example, handles the same issue with *batch verification* [112, §4]: the evaluations, and the random values that feed into them, are reused over parallel instances of the proof protocol (on different inputs). But batch verification, in our context, would require $\mathcal{P}$ to store intermediate gate values (§3.2) for the entire batch.

[8] This channel is not covert: failures are noticeable to the operator. However, if the operator expects some random hardware faults, cheating chips may try to hide a low-bandwidth communication channel in such faults.

cost	verifier	prover
energy		
compute	$(7d\log G+6G)\,E_{\text{mul,t}}+(15d\log G+2G)\,E_{\text{add,t}}$	$dG\log^2 G\cdot E_{\text{mul,u}}+9dG\log G\cdot E_{\text{add,u}}+4dG\log G\left\langle E_{\text{g,u}}\right\rangle$
$\mathcal{V}$-$\mathcal{P}$ *tx*	$(2d\log G+G)\,E_{\text{tx,t}}$	$(7d\log G+G)\,E_{\text{tx,u}}$
store	$d\log G\cdot E_{\text{sto,t}}$	$dG\cdot n_{\mathcal{P},\text{pl}}\cdot E_{\text{sto,u}}$
PRNG	$2d\log G\cdot E_{\text{prng,t}}$	—
$\mathcal{V}$ *I/O*	$2G\cdot E_{\text{io,t}}$	—
area		
compute	$n_{\mathcal{V},\text{sc}}\left(2A_{\text{mul,t}}+3A_{\text{add,t}}\right)+2n_{\mathcal{V},\text{io}}\left(A_{\text{mul,t}}+A_{\text{add,t}}\right)$	$n_{\mathcal{P},\text{sc}}\left(7G\cdot A_{\text{mul,u}}+\lfloor 7G/2\rfloor\cdot A_{\text{add,u}}\right)$
$\mathcal{V}$-$\mathcal{P}$ *tx*	$(2d\log G+G)\,A_{\text{tx,t}}$	$(7d\log G+G)\,A_{\text{tx,u}}$
store	$d\log G\cdot A_{\text{sto,t}}$	$dG\cdot n_{\mathcal{P},\text{pl}}\cdot A_{\text{sto,u}}$
PRNG	$2d\log G\cdot A_{\text{prng,t}}$	—
$\mathcal{V}$ *I/O*	$2G\cdot A_{\text{io,t}}$	—
delay: Zebra's overall throughput is $1/\max(\mathcal{P}\text{ delay},\mathcal{V}\text{ delay})$; the expressions for $\mathcal{P}$ and $\mathcal{V}$ delay are given immediately below:		
	$\max\left\{d/n_{\mathcal{V},\text{sc}}\left(2\log G\left(\lambda_{\text{mul,t}}+2\lambda_{\text{add,t}}\right)+\lceil(7+\log G)/2\rceil\,\lambda_{\text{mul,t}}+4\lambda_{\text{add,t}}\right),\right.$ $\left.\left(\lceil 3G/n_{\mathcal{V},\text{io}}+\log n_{\mathcal{V},\text{io}}\rceil\right)\lambda_{\text{mul,t}}+\lceil\log n_{\mathcal{V},\text{io}}\rceil\,\lambda_{\text{add,t}}\right\}$	$d/n_{\mathcal{P},\text{sc}}\left[3\log^2 G\cdot\lambda_{\text{add,u}}+18\log G\left(\lambda_{\text{mul,u}}+\lambda_{\text{add,u}}\right)\right]$

$n_{\mathcal{V},\text{io}}$: $\mathcal{V}$ parameter; trades area vs i/o delay
$n_{\mathcal{V},\text{sc}}$: $\mathcal{V}$ parameter; trades area vs sumcheck delay
$n_{\mathcal{P},\text{pl}}$: $\mathcal{P}$ parameter; # in-flight runs
$n_{\mathcal{P},\text{sc}}$: $\mathcal{P}$ parameter; trades area vs delay
$\left\langle E_{\text{g,u}}\right\rangle$: mean per-gate energy of $\mathcal{C}$, untrusted
d, G: depth and width of arithmetic circuit $\mathcal{C}$
$E_{\{\text{add,mul,tx,sto,prng,io}\},\{\text{t,u}\}}$: energy cost in {trusted, untrusted} technology node for {+, ×, $\mathcal{V}$-$\mathcal{P}$ interaction, store, PRNG, $\mathcal{V}$ I/O}
$A_{\{\text{add,mul,tx,sto,prng,io}\},\{\text{t,u}\}}$: area cost in {trusted, untrusted} technology node for {+, ×, $\mathcal{V}$-$\mathcal{P}$ interaction, store, PRNG, $\mathcal{V}$ I/O}
$\lambda_{\{\text{add,mul}\},\{\text{t,u}\}}$: delay in {trusted, untrusted} technology node for {+, ×}

FIGURE 5—$\mathcal{V}$ and $\mathcal{P}$ costs as a function of $\mathcal{C}$ parameters and technology nodes (simplified model; low-order terms discarded). We assume $|x|=|y|=G$. Energy and area constants for interaction, store, PRNG, and I/O indicate costs for a single element of $\mathbb{F}_p$. $\mathcal{V}$-$\mathcal{P}$ tx is the cost of interaction between $\mathcal{V}$ and $\mathcal{P}$; $\mathcal{V}$ I/O is the cost for the operator to communicate with Zebra (§4). For $\mathcal{P}$, store is the cost of buffering pipelined computations (§3.2); for $\mathcal{V}$, it is the cost to retrieve and decrypt auxiliary inputs (§4). Transmit, store, and PRNG occur in parallel with execution, so their delay is not included, provided that the corresponding circuits execute quickly enough (§5, §7.1). Physical quantities depend on both technology node and implementation particulars; we give values for these quantities in Section 7.1.

ing multilinear extensions of x and y, respectively (§3.5). In our evaluation we constrain these parameters to keep area at or below some fixed size, for manufacturability (§2.3).

Storage versus precomputation. With regard to $\mathcal{V}$'s auxiliary inputs, the cost model accounts for retrieving and decrypting (§4); the cost of precomputing, and its amortization, was covered in Section 4.

6 Implementation of Zebra

Our implementation of Zebra comprises four components.

The first is a compiler toolchain that produces $\mathcal{P}$. The toolchain takes as input a high-level description of an AC (in an intermediate representation emitted by the Allspice compiler [1]) and designer-supplied primitive blocks for field addition and multiplication in $\mathbb{F}_p$; the toolchain produces a *synthesizable* SystemVerilog implementation of Zebra. The compiler is written in C++, Perl, and SystemVerilog (making heavy use of the latter's metaprogramming facilities [7]).

Second, Zebra contains two implementations of $\mathcal{V}$. The first is a parameterized implementation of $\mathcal{V}$ in SystemVerilog. As with $\mathcal{P}$, the designer supplies primitives for field arithmetic blocks in $\mathbb{F}_p$. Additionally, the designer selects the parameters $n_{\mathcal{V},\text{sc}}$ and $n_{\mathcal{V},\text{io}}$ (§5, §7.4). The second is a software implementation adapted from Allspice's verifier [1].

Third, Zebra contains a C/C++ library that implements $\mathcal{V}$'s input-independent precomputation for a given AC, using the same high-level description as the toolchain for $\mathcal{P}$.

Finally, Zebra implements a framework for cycle-accurate RTL simulations of complete interactions between $\mathcal{V}$ and $\mathcal{P}$. It does so by extending standard RTL simulators. The extension is an interface that abstracts communication between $\mathcal{P}$ and $\mathcal{V}$ (§2.2), and between $\mathcal{V}$ and the store of auxiliary inputs (§4). This framework supports both the hardware and software implementations of $\mathcal{V}$. The interface is written in C using the Verilog Procedural Interface (VPI) [7], and was tested with the Cadence Incisive [3] and Icarus Verilog [6] RTL simulators.

In total, our implementation comprises approximately 6000 lines of SystemVerilog, 10400 lines of C/C++ (partially inherited from Allspice [1]), 600 lines of Perl, and 600 lines of miscellaneous scripting glue.

7 Evaluation

This section answers: *For which ACs can Zebra outperform the native baseline?* Section 8 makes this evaluation concrete by comparing Zebra to real-world baselines for specific applications. In sum, Zebra wins in a fairly narrow regime: when there is a large technology gap and the computation is large. These conclusions are based on the cost model (§5), recent figures from the literature, standard CMOS scaling models, and dozens of synthesis- and simulation-based measurements.

Method. Zebra's implementation (§6) allows us to evaluate using both synthesis experiments and cycle-accurate simulations. However, this approach comes with a practical limitation: synthesizing and simulating even a moderately sized chip design can take thousands of core-hours; meanwhile the aim

	350 nm ($\mathscr{V}$)		45 nm ($\mathscr{P}$)	
	$\mathbb{F}_p + \mathbb{F}_p$	$\mathbb{F}_p \times \mathbb{F}_p$	$\mathbb{F}_p + \mathbb{F}_p$	$\mathbb{F}_p \times \mathbb{F}_p$
energy (nJ/op)	3.1	220	0.006	0.21
area (μm^2)	2.1×10^5	27×10^5	6.5×10^3	69×10^3
delay (ns)	6.2	26	0.7	2.3

FIGURE 6—Synthesis data for field operations in $\mathbb{F}_p$, $p = 2^{61} - 1$.

	350 nm ($\mathscr{V}$)		45 nm ($\mathscr{P}$)	
	pJ/$\mathbb{F}_p$	μm^2/($\mathbb{F}_p$/ns)	pJ/$\mathbb{F}_p$	μm^2/$\mathbb{F}_p$
$\mathscr{V}$-$\mathscr{P}$ **tx**	1100	3400	600	$1900 \cdot$ ns
store	48×10^3	380×10^6	4.2	80
PRNG	8700	17×10^6	—	—
$\mathscr{V}$ **I/O**	4200	2×10^6	—	—

FIGURE 7—Costs for communication, storage (including decryption, for $\mathscr{V}$; §4), and PRNG; extrapolated from published results [27, 66, 68, 75, 79, 87, 89, 93, 97, 109] using standard scaling models [67].

here is to characterize Zebra over a wide range of ACs. Thus, we leverage the analytical cost model described in Section 5. We do so in three steps.

First, we obtain values for the parameters of this model by combining synthesis results with data published in the literature (§7.1). Second, we validate the cost model by comparing predictions from the model with both synthesis results and cycle-accurate simulations; these data closely match the analytical predictions (§7.2). Third, we estimate the baseline's costs (§7.3) and then use the validated cost model to measure Zebra's performance relative to the baseline, across a range of arithmetic circuit and physical parameters (§7.4).

7.1 Estimating cost model parameters

In this section, we estimate the energy, area, and delay of Zebra's basic operations (§5, (1)–(6)).

Synthesis of field operations. Figure 6 reports synthesis data for both field operations in $\mathbb{F}_p$, $p = 2^{61} - 1$, which admits an efficient and straightforward modular reduction. Both operations were written in Verilog and synthesized to two technology libraries: Nangate 45 nm Open Cell [8] and a 350 nm library from NC State University and the University of Utah [9]. For synthesis, we use Cadence Encounter RTL Compiler [3].

Communication, storage, and PRNG costs. Figure 7 reports area and energy costs of communication, storage, and random number generation using published measurements from built chips. Specifically, we use results from CPUs built with 3D packaging [75, 79], SRAM designs [89], solid-state storage [53, 93], ASICs for cryptographic operations [66, 68, 109], and optical interconnects [27, 87, 97].

For all parameters, we use standard CMOS scaling models to extrapolate to other technology nodes for evaluation.[9]

[9] A standard technique in CMOS circuit design is projecting how circuits will scale into other technology nodes. Modeling this behavior is of great practical interest, because it allows accurate cost modeling prior to designing and fabricating a chip. As a result, such models are regularly used in industry [67].

	$\log G$	measured	predicted	error
area	4	8.76	9.42	+7.6%
(mm^2)	5	17.06	18.57	+8.8%
	6	33.87	36.78	+8.6%
	7	66.07	73.11	+11%
delay	4	682	681	-0.2%
(cycles)	5	896	891	-0.6%
	6	1114	1115	+0.1%
	7	1358	1353	-0.4%
+,× **ops**	4	901, 1204	901, 1204	0%
	5	2244, 3173	2244, 3173	0%
	6	5367, 8006	5367, 8006	0%
	7	12494, 19591	12494, 19591	0%

FIGURE 8—Comparison of cost model (§5) with measured data. Area numbers come from synthesis; delay and operation counts come from cycle-accurate RTL simulation.

7.2 Validating the cost model

We use synthesis and simulation to validate the cost model (§5). To measure area, we synthesize sub-provers (§3.1) for AC layers over several values of G. To measure energy and throughput, we perform cycle-accurate RTL simulations for the same sub-provers, recording the pipeline delay (§5, Fig. 5) and the number of invocations of field addition and multiplication.[10]

Figure 8 compares our model's predictions to the data obtained from synthesis and simulation. The model predicts slightly greater area than the synthesis results show. This is likely because, in the context of a larger circuit, the synthesizer has more information (e.g., related to critical timing paths) and thus is better able to optimize the field arithmetic primitives.

Although we have not presented our validation of $\mathscr{V}$'s costs, its cost model has similar fidelity.

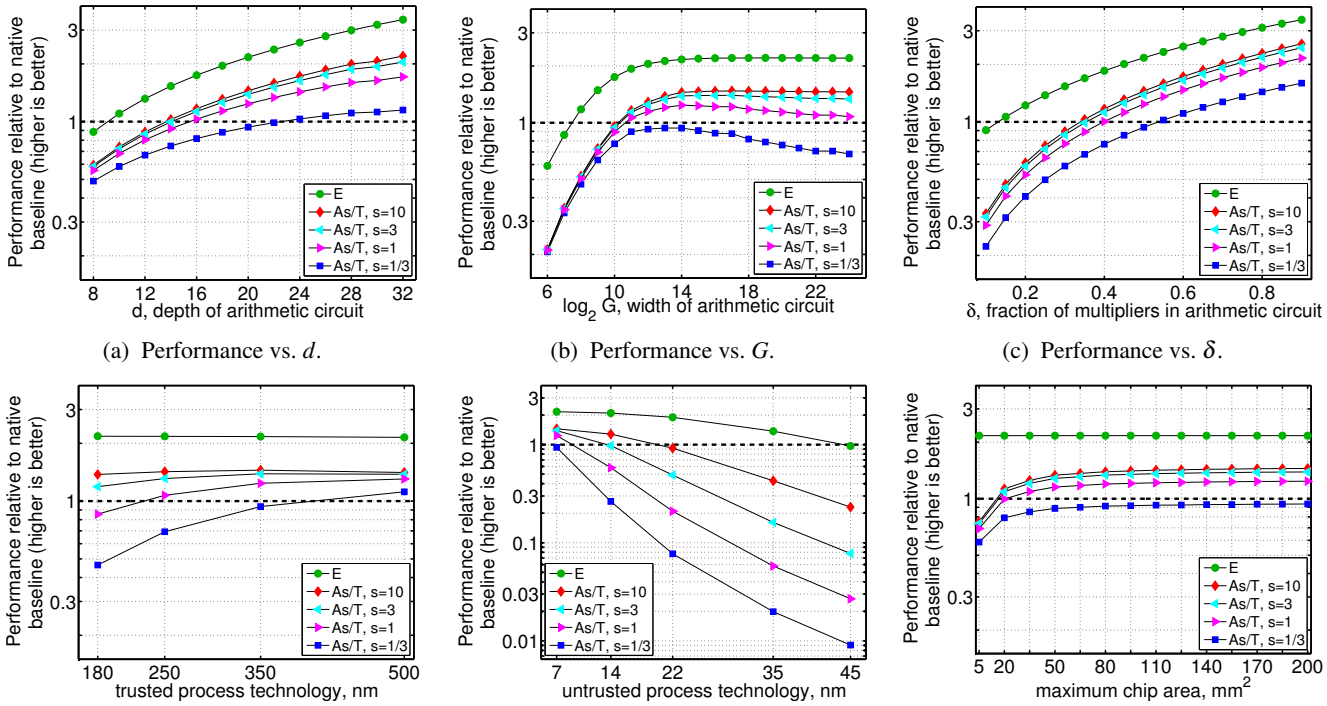
7.3 Baseline: native trusted implementations

For an arithmetic circuit with depth d, width G, and a fraction δ of multiplication gates, we estimate the costs of directly executing the AC in the trusted technology node. To do so, we devise an optimization procedure that minimizes A/T (under the constraint that total area is limited to some $A_{\max}$; §2.3). Since this baseline is a direct implementation of the arithmetic circuit, we account E as the sum of the energy for each operation in the AC, plus the energy for I/O (§4).

Optimization proceeds in two steps. In the first step, the procedure apportions area to multiplication and addition primitives, based on δ and on the relative time and area cost of each operation. In the second step, the procedure chooses a pipelining strategy that minimizes delay, subject to sequencing requirements imposed by $\mathscr{C}$'s layering.

It is possible, through hand optimization, to exploit the structure of particular ACs in order to improve upon this op-

[10] We use the number of field operations as a proxy for the energy consumed by $\mathscr{P}$'s sub-provers in executing the protocol. This gives a good estimate because (1) the sub-prover's area is dominated by these circuits (Fig. 8), and (2) the remaining area of each sub-prover is dedicated to control logic, which includes only slowly-varying signals with negligible energy cost.



(a) Performance vs. d. (b) Performance vs. G. (c) Performance vs. δ.

(d) Performance vs. trusted node. (e) Performance vs. untrusted node. (f) Performance vs. maximum chip area.

FIGURE 9—Zebra performance relative to baseline (§7.3) on E and A_s/T metrics (§2.3), varying $\mathscr{C}$ parameters, technology nodes, and maximum chip area. In each case, we vary one parameter and fix the rest. Fixed parameters are: trusted technology node = 350 nm; untrusted technology node = 7 nm; depth of $\mathscr{C}$, $d = 20$; width of $\mathscr{C}$, $G = 2^{14}$; fraction of multipliers in $\mathscr{C}$, $\delta = 0.5$; maximum chip area, $A_{\max} = 200$ mm^2; maximum power dissipation, $P_{\max} = 150$ W. In all cases, designs follow the optimization procedures described in Sections 7.1–7.3.

timization procedure, but our goal is a procedure that gives good results for generic ACs. We note that our optimization procedure is realistic: it is roughly similar to the one used by automated hardware design toolkits such as Spiral [84].

7.4 Zebra versus baseline

This section evaluates the performance of Zebra versus the baseline, on the metrics E and A_s/T, as a function of $\mathscr{C}$ parameters (width G, depth d, fraction of multiplication gates δ), technology nodes, and maximum allowed chip size. We vary each of these parameters, one at a time, fixing others. Fixed parameters are as follows: $d = 20$, $G = 2^{14}$, $\delta = 0.5$, trusted technology node = 350 nm, untrusted technology node = 7 nm, and $A_{\max} = 200$ mm^2. We limit Zebra to no more than $P_{\max} = 150$ W, which is comparable to the power dissipation of modern GPUs [2, 5], and we vary $s \in \{1/3, 1, 3, 10\}$ (§2.3).

For each design point, we fix $\mathscr{V}$'s area equal to the native baseline area, and set $n_{\mathscr{P},\text{pl}} = d$. We then optimize $n_{\mathscr{P},\text{sc}}$, $n_{\mathscr{V},\text{io}}$, and $n_{\mathscr{V},\text{sc}}$ (§5, Fig. 5). To do so, we first set $n_{\mathscr{V},\text{io}}$ and $n_{\mathscr{V},\text{sc}}$ to balance delay among $\mathscr{V}$'s layers (§3.5), and to minimize these delays subject to area limitations. We choose $n_{\mathscr{P},\text{sc}}$ so that $\mathscr{P}$'s pipeline delay is less than or equal to $\mathscr{V}$'s.

Figure 9 summarizes the results. In each plot, the break-even point is designated by the dashed line at 1. We observe the following trends:

- As $\mathscr{C}$ grows in size (Figs. 9a, 9b) or complexity (Fig. 9c), Zebra's performance improves compared to the baseline.
- As the performance gap between the trusted and untrusted technology nodes grows (Figs. 9d, 9e), Zebra becomes increasingly competitive with the baseline, in both E and A_s/T.
- As G grows, $\mathscr{P}$'s area increases (§5, Fig. 5), making A_s/T worse even as E improves (Fig. 9b). This is evident in the $s = 1/3$ curve, which is highly sensitive to $\mathscr{P}$'s area.
- Finally, we note that Zebra's competitiveness with the baseline is relatively insensitive to maximum chip area (Fig. 9f).

7.5 Summary

We have learned several things:

Zebra beats the baseline in a narrow but distinct regime. In particular, Zebra requires more than a decade's technology gap between $\mathscr{V}$ and $\mathscr{P}$. Zebra also requires that the computation is "hard" for the baseline, i.e., it involves tens of thousands of operations (Figs. 9a, 9b), with thousands of expensive operations per layer (Figs. 9b, 9c). At a high level, the technology gap offsets $\mathscr{P}$'s proving costs, while "hard" computations allow $\mathscr{V}$'s savings to surpass the overhead of the protocol (§2.2).

The price of verifiability is high. Even when Zebra breaks even, it is orders of magnitude more costly than executing Ψ on an untrusted technology node.

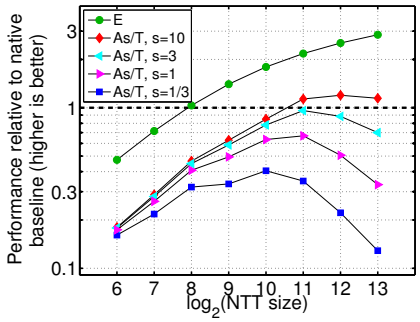


FIGURE 10—Zebra performance relative to baseline (§8.1) on E and A_s/T metrics (§2.3), for NTT with a given number of points, over $\mathbb{F}_p$, $p = 2^{63}+2^{19}+1$. Untrusted technology node = 350 nm; trusted technology node = 7 nm; maximum chip area, $A_{\max} = 200$ mm^2.

But "there are no hidden fees" (we hope). Although the stated price is high, we have endeavored to account for all of the costs that we could think of. Thus, when a computation falls into Zebra's break-even zone—as with the examples in the next section—we can, with reasonable confidence, claim a genuine win.

8 Applications

This section evaluates Zebra on two concrete computations: the number theoretic transform over $\mathbb{F}_p$, and Curve25519 point multiplication.

8.1 Number theoretic transform over $\mathbb{F}_p$

The number theoretic transform (NTT) is a linear transform closely related to the FFT. Its input is $x = \{a_0, \ldots, a_{N-1}\}$, the coefficients of a degree $N-1$ polynomial $p(t)$; its output is $y = \{p(\omega^0), \ldots, p(\omega^{N-1})\}$, where ω is a primitive N^{th} root of unity in $\mathbb{F}_p$. This transform is widely used in signal processing, computer algebra packages [103], and even in some cryptographic algorithms [20].

The fastest known algorithm is iterative, and uses a sequence of *butterfly operations* [54, Ch. 30]. A butterfly takes as input $x_1, x_2 \in \mathbb{F}_p$, and outputs $y_1 = \omega^i (x_1 + x_2)$ and $y_2 = \omega^j (x_1 - x_2)$ for some $i, j \in \{0, \ldots, N-1\}$.

Implementing NTT in Zebra

The iterative NTT is essentially an arithmetic circuit. However, if implemented naively in Zebra, that AC would introduce overhead. The core issue is that, if the AC uses add and multiply gates, then the $x_1 - x_2$ term in a butterfly would be computed as $-1 \times x_2$, followed by an addition in the next layer.

To avoid increasing the depth of the AC, we take inspiration from CMT's enhanced gates [55, §3.2] and introduce *subtraction gates*. For $\mathscr{P}$, this change implies a new type of gate prover (§3.1). For $\mathscr{V}$, we define a new wiring predicate (§2.2), sub_i, and its multilinear extension, $\tilde{\text{sub}}_i$; at the end of each sumcheck invocation (line 100, Fig. 2), $\mathscr{V}$ adds the term $\tilde{\text{sub}}_i(q_{i-1}, w_0, w_1)(h_0 - h_1)$. Accordingly, an auxiliary input includes d more precomputed values (§2.2, §4).

	$N = 2^6$		$N = 2^{10}$		$N = 2^{13}$	
cost	$\mathscr{V}$	$\mathscr{P}$	$\mathscr{V}$	$\mathscr{P}$	$\mathscr{V}$	$\mathscr{P}$
energy, μJ						
compute	370	0.21	2700	10	17 000	150
$\mathscr{V}$-$\mathscr{P}$ *tx*	0.29	0.42	1.7	1.6	10	6.7
store	8.6	0.11	19	4.8	29	65
PRNG	1.7	—	4.3	—	7	—
$\mathscr{V}$ *I/O*	0.58	—	8.9	—	71	—
area, mm^2						
compute	44	2.8	51	22	51	175
$\mathscr{V}$-$\mathscr{P}$ *tx*	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1
store	24	< 0.1	9	1.6	2	22
PRNG	1.3	—	0.5	—	0.1	—
$\mathscr{V}$ *I/O*	0.2	—	0.3	—	0.3	—
delay, μs	3.1	3.1	20	18	133	32.9
$n_{\{\mathscr{V},\mathscr{P}\},\text{sc}}$	4	2	2	1	1	1
$n_{\mathscr{V},\text{io}}$	3	—	6	—	7	—
$n_{\mathscr{P},\text{pl}}$	—	12	—	20	—	26
total power, W	121		140		133	

FIGURE 11—Costs for N-point NTT (§8.1), $N \in \{2^6, 2^{10}, 2^{13}\}$.

Another issue is that computing a length-N NTT requires $N/2$ powers of ω; thus, an implementation must either receive these values as inputs or compute them directly. Because $\mathscr{V}$'s costs scale linearly with the number of inputs but only logarithmically with the size of each layer of $\mathscr{C}$ (§2.2, §5), computing powers of ω in $\mathscr{C}$ is the natural approach—but doing so naively at least doubles the depth of $\mathscr{C}$, erasing any savings. Instead, Zebra's $\mathscr{C}$ takes as input $\log N$ values, $\{\omega, \omega^2, \omega^4, \omega^8, \ldots\}$, and $\mathscr{C}$ computes other powers of ω just as they are needed. This approach reduces costs by 25% or more compared to taking powers of ω as inputs, and it adds no additional layers to the iterative NTT's arithmetic circuit.

Evaluation

Baseline. The baseline is an arithmetic circuit implementing the iterative algorithm for the NTT, optimized as described in Section 7.3. We make the conservative assumption that computing powers of ω is free for the baseline.

Method. We evaluate Zebra versus the baseline on the metrics E and A_s/T, for length-N NTTs over $\mathbb{F}_p$, $p = 2^{63}+2^{19}+1$.[11] As in Section 7.4, we fix trusted technology node = 350 nm, untrusted technology node = 7nm, $A_{\max} = 200$ mm^2, $P_{\max}$=150 W, $n_{\mathscr{P},\text{pl}} = d$ (§5). We vary $s \in \{1/3, 1, 3, 10\}$, and $\log N \in \{6, \ldots, 13\}$.

We generate Verilog for $\mathscr{P}$ and $\mathscr{V}$ using our compiler toolchain and run cycle-accurate RTL simulations for sizes up to $\log N = 6$ (§6). We use these simulations to confirm that the cost model is calibrated for the new field (§5, §7.2), and then use the model to compute costs for larger N.

Comparison with baseline. Figure 10 depicts Zebra's performance relative to the baseline. As in Section 7.4, Zebra is not competitive on E for small computations, but it equals or

[11] We use a different p from §7.1 because the NTT requires a field with a subgroup of size N. This is satisfied by p for $\log N \leq 19$, N a power of 2.

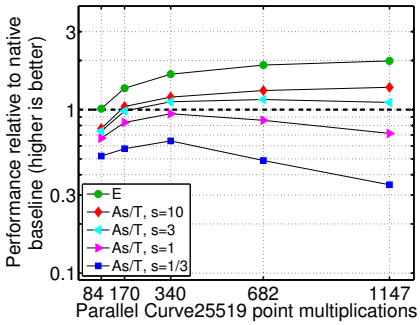


FIGURE 12—Zebra performance relative to baseline (§8.2) on E and A_s/T metrics (§2.3), versus number of parallel Curve25519 evaluations. Untrusted technology node = 350 nm; trusted technology node = 7 nm; maximum chip area, $A_{\max} = 200$ mm^2.

exceeds the baseline when N is large enough; this is consistent with $\mathscr{V}$'s asymptotic efficiency (§2.2). Relative A_s/T lags E because the native computation is only about 30% multiplications (recall that Zebra's performance suffers when there are few multiplications; §7.4). Further, for large NTTs, $\mathscr{P}$'s area grows large; thus A_s grows, and A_s/T worsens.

Concrete costs. Figure 11 tabulates concrete costs for three NTT designs: $N = 2^6$, $N = 2^{10}$, and $N = 2^{13}$. In all three cases, most of the energy cost is in $\mathscr{V}$'s computation, because basic operations are much more expensive in the trusted technology node. In fact, this energy cost is high enough that it limits $\mathscr{V}$'s throughput: it would be possible to add more compute hardware to $\mathscr{V}$ (since its area is less than the maximum), but doing so would increase T and thus exceed the power budget. As N increases, $\mathscr{V}$'s throughput decreases, reducing the area required for store and PRNG circuits (§4, §7.1). $\mathscr{P}$'s area requirement increases with N because the size of sub-provers increases and because the required storage for pipelining in-flight runs increases (§3.2).

8.2 Curve25519 point multiplication

Curve25519 is a high-performance elliptic curve used in many cryptographic protocols [14, 34]. An operation of interest for hardware acceleration is *point multiplication* [21, Ch. 13]. This operation takes as inputs a 255-bit scalar value v and an elliptic curve point Q, and computes the point $R = [v]Q$ via a sequence of 255 *double-and-add* steps [21], one for each bit of v. (For efficiency and security, point multiplication uses a Montgomery ladder [21, 34, 85].)

Efficient point multiplication in Zebra

Double-and-add is naturally expressed as an arithmetic circuit over $\mathbb{F}_p$, $p = 2^{255} - 19$, with depth $d = 7$ and width $G \approx 8$. Thus, a full point multiplication (255 double-and-add steps) is deep and narrow, both problems for Zebra (§7.4).

Thus, Zebra reformulates the AC, turning the long sequence of double-and-add steps into shorter sequences. Specifically, under Zebra, the depth of the AC is given by 5 double-and-add steps, and computing a point multiplication requires

	$K = 170$		$K = 682$		$K = 1147$	
cost	$\mathscr{V}$	$\mathscr{P}$	$\mathscr{V}$	$\mathscr{P}$	$\mathscr{V}$	$\mathscr{P}$
energy, μJ						
compute	32 000	250	92 000	1300	145 000	2500
$\mathscr{V}$-$\mathscr{P}$ *tx*	6	8	16	14	25	19
store	120	4	140	15	150	26
PRNG	27	—	32	—	35	—
$\mathscr{V}$ *I/O*	24	—	95	—	160	—
area, mm^2						
compute	194	55	192	110	175	185
$\mathscr{V}$-$\mathscr{P}$ *tx*	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1	< 0.1
store	4.3	2.2	1.5	9	0.9	15
PRNG	0.3	—	< 0.1	—	< 0.1	—
$\mathscr{V}$ *I/O*	< 0.1	—	< 0.1	—	< 0.1	—
delay, μs	260	190	840	450	1400	490
$n_{\{\mathscr{V},\mathscr{P}\},\text{sc}}$	4	2	2	1	1	1
$n_{\mathscr{V},\text{io}}$	9	—	11	—	11	—
$n_{\mathscr{P},\text{pl}}$	—	35	—	35	—	35
total power, W	123		111		105	

FIGURE 13—Costs for Curve25519 (§8.2), for $K = 170$, $K = 682$, and $K = 1147$ parallel sequences of 5 double-and-add operations.

51 ($= 255/5$) iterated protocol runs; the cost of this change is that $\mathscr{V}$ handles inputs and outputs 51 times. And for width, the AC contains many parallel double-and-add sequences.

Another issue with the double-and-add arithmetic circuit is that it uses a "conditional swap" operation, interchanging two values based on one bit of the scalar v. Conditional swap can be implemented with only $+$ and $\times$, but as with subtraction (§8.1), a more efficient alternative is to introduce a new type of gate, a multiplexer or *mux*, which selects either its left or right input based on a *selector bit*. The details of mux gates are described in the extended version of this paper [113].

For efficiency, we require that all of the mux gates in any layer of an arithmetic circuit share one selector bit. For point multiplication, this means that all parallel double-and-add operations in a protocol run must use the same scalar v. This restriction is acceptable, for example, for TLS servers using ECDH_ECDSA key exchange [38], or whose ECDHE implementations cache and reuse ephemeral keys for many clients [11; 48, §4.2].

Evaluation

Baseline. Consistent with existing Curve25519 hardware implementations [95, 96], the baseline directly executes a sequence of 255 double-and-add steps.

Method. We compare Zebra and the baseline on E and A_s/T, for designs that compute K parallel sequences of double-and-add, $K \in \{84, 170, 340, 682, 1147\}$. The first four values give $\lceil \log G \rceil \in \{7, 8, 9, 10\}$; $K = 1147$ is the largest for which $\mathscr{P}$ fits into $A_{\max}$. We fix other parameters as in Section 8.1.

As in Sections 7 and 8.1, we synthesize field addition and multiplication primitives, and use cycle-accurate Verilog simulations to check the cost model's calibration (§5, §7.2). We then use the model to compute costs for large K.

Comparison with baseline. Figure 12 depicts Zebra's performance relative to the baseline. As in previous experiments, Zebra is competitive on E for large computations. Like NTT, double-and-add is only about 30% multiplication gates; together with the area overhead due to $\mathscr{P}$, Zebra is less competitive on A_s/T, especially for small s.

Concrete costs. Figure 13 tabulates concrete costs for $K = 170$, $K = 682$, and $K = 1147$. The costs are given for a run of 5 double-and-add operations; a full point multiplication entails 51 runs. As in the NTT, $\mathscr{V}$ dominates energy cost. However, in this application, $\mathscr{V}$'s performance is limited by area rather than power dissipation because of the design of the primitive circuits for $p = 2^{255} - 19$ field operations.

9 Limitations and discussion

Zebra has clear limitations: the manufacturing and operational requirements are unusual (§4); the overhead is high, compared to untrusted execution; and, although some real computations fall within its regime of applicability (§8), that regime is narrow (§7). Below, we contextualize these limitations, and then address some loose ends in the cost analysis.

Zebra's applicability in context. Recall that, even before we considered Zebra's break-even zone (§7), we narrowed Zebra's focus to computations Ψ that have natural expressions as ACs (§2.2). What would happen if Ψ were a Boolean circuit, or a normal program over fixed-width integers? Then, the native version would run extremely efficiently on an ASIC or CPU, so getting $\mathscr{V}$'s $O(d \cdot \log G)$ running time to be concretely lower would require an enormous G—in other words, an unrealistically large number of parallel instances.

This limitation might seem overly restrictive, but something similar applies to every built system for verifiable outsourcing [24, 25, 29, 31–33, 41, 51, 55, 56, 58, 59, 61, 76, 88, 98–101, 106, 108, 112, 114]. As an example, the state-of-the-art SNARK [36, 61, 88] implementation is `libsnark` [17, 33], which is an optimized implementation of the GGPR-based [61] SNARK in Pinocchio [88]. Disregarding precomputation and the cost to handle inputs and outputs, `libsnark`'s minimum verification time is 6 ms on a 2.7 GHz CPU [114, §2.4, Fig. 3, §5.3]. For a native computation to take longer than 6 ms on the same CPU requires that it perform more than 16 million operations. But the maximum problem size that `libsnark` can handle (on a machine with 32 GB of memory) is 10 to 16 million operations [33, 114], so breaking even is nearly impossible unless the operations in the original computation were non-native; $\mathbb{F}_p$ operations work best.

(Some systems report beating native execution on 110×110 matrix multiplication [88], but our experiments—which use a similar CPU (an Intel Core i5-4440), normal compiler flags (`-O3`), and standard environments—yield a sub-millisecond native cost for this computation, which is an order of magnitude lower than the reported verification time.)

In theory, these issues are surmountable. For example, the scaling limit on GGPR-based SNARKs can be circumvented by composition [32, 37, 56]. And, whereas OptimizedCMT is limited to ACs that are deterministic and wide, GGPR-based systems handle a much broader class of computations and offer far more properties (§10). But implementing GGPR in hardware proved to be a bottleneck, as we describe next.

Experience with other verifiable outsourcing machinery. We began this project by studying machinery based on GGPR [61], specifically the Zaatar [99] interactive argument and the Pinocchio [88] SNARK (§10). However, despite several months of work, we were unable to produce a physically realizable design that achieved reasonable performance.

First, we found it difficult to extract sufficient parallelism. In Zebra, sub-provers can be arranged in a pipeline because OptimizedCMT's interaction proceeds layer-by-layer, with narrow dependencies (§3.2). In contrast, GGPR's proofs are computed in stages, but these stages operate over the entire execution of $\mathscr{C}$; this made pipelining both difficult and ineffective.

We also had trouble exploiting locality. As an example, both OptimizedCMT and GGPR require the prover to evaluate polynomials. In OptimizedCMT, these polynomials are small (total degree at most $O(\log G)$; §2.2), and importantly, the evaluations can be computed incrementally (§2.2, §3.3). Meanwhile, GGPR's prover both evaluates and interpolates polynomials that are exponentially larger (total degree $O(d \cdot G)$).

Finally, we found it difficult to reuse modules in GGPR. In Zebra this is possible because the prover's work is just field operations (§3.4). GGPR, on the other hand, has greater diversity of operations—FFTs and cryptographic primitives—which have vastly different control and data flow.

There remain other details of GGPR that we did not even begin to address—for example, how the prover might handle a common reference string with tens of millions of field elements. Of course, this does not imply that an efficient hardware implementation of GGPR is fundamentally out of the question, and we hope that future work will produce one!

When CPUs are trusted. In evaluating Zebra, we did not consider the natural baseline of executing Ψ on a CPU that the principal trusts. This option has the usual advantages of software: simplicity, flexibility, no non-recurring costs (see below), ease of programming, etc. If the principal trusts a CPU in a state-of-the-art technology node, or if expressing Ψ as an arithmetic circuit entails significant overhead (§2.2), executing Ψ directly on a trusted CPU is likely the best option. However, the answer is less clear when the trusted CPU is in a mature technology node and Ψ is naturally expressed as an AC. Since this is Zebra's approximate applicability regime, meaning that Zebra outperforms a native ASIC in the trusted technology node, Zebra most likely outperforms a CPU in that same technology node (because ASICs far outperform general-purpose CPUs).

Other alternative baselines exist; for example, execution on an untrusted ASIC coupled with auditing or with later re-

execution on a trusted processor. These approaches may be appropriate in some settings; we leave their investigation and evaluation to future work.

Non-recurring costs. The major non-recurring costs when building ASICs are engineering effort and photolithographic masks. To briefly compare the baseline and Zebra, both need designs and implementations—of Ψ and $\mathcal{V}$, respectively—that always work and have no defects. While we do not know the exact costs of producing such *golden implementations*—this is the province of trusted foundries [15, 45], which require trusted designers, trusted mask productions, etc.—we are sure that it isn't cheap.

On top of this, Zebra incurs costs for the prover that are likely to be high: in the latest technology nodes, a full mask set costs several million dollars. On the other hand, Zebra has an advantage that the baseline does not: the work to create $\mathcal{V}$ amortizes over many computations, as explained shortly. Thus, relative non-recurring costs roughly boil down to the baseline's golden Ψ versus Zebra's untrusted $\mathcal{P}$. We do not know which one this comparison favors. We note that if the golden Ψ is much more expensive, then one may wish to use Zebra, *even if it performs worse on the metrics evaluated earlier*.

Using one $\mathcal{V}$ for different computations. The non-recurring costs of producing $\mathcal{V}$ amortize over multiple computations because the computation that $\mathcal{V}$ is verifying is configured *after* manufacture. Specifically, the arithmetic circuit, $\mathcal{C}$, appears in $\mathcal{V}$'s workload only through the precomputed auxiliary inputs: aside from line 100 in Figure 2, $\mathcal{V}$'s algorithm is independent of $\mathcal{C}$. (In fact, a slightly modified design for $\mathcal{V}$ might even accept d, G, $|x|$, and $|y|$ as run-time parameters.) As a result, $\mathcal{V}$ has to be designed only once, or a small number of times with different design parameters (§3.5,§5).

10 Related work

Zebra relates to two strands of work: defenses against hardware Trojans and built systems for verifiable outsourcing.

Hardware Trojans. Tehranipoor and Koushanfar [105] have surveyed the threat of hardware Trojans; Bhunia et al. [35] have surveyed defenses. We follow the taxonomy of Bhunia et al., in which defenses are classified as *post-fabrication* detection, *run-time* monitoring, or *design-time* deterrence.

Post-fabrication is sub-divided into *logic testing*, *side-channel* analysis, and *golden reference*. *Logic testing* works within existing IC testing frameworks: these techniques exercise the chip with certain inputs and verify that the outputs are correct [42]. Wolff et al. [120] and Chakraborty et al. [47] augment these tests with certain sequences of inputs (or "cheat codes"), that are likely to trigger a Trojan. However, these techniques do not provide comprehensive guarantees (untested inputs could still produce incorrect outputs) or defend against Trojans designed to stay dormant during post-fabrication testing [105, 115] (for instance, those activated by internal timers).

Side-channel analysis aims to detect changes in the measured delay and power of chips versus estimates obtained from pre-fabrication simulations [26, 73, 77, 78, 119]. However, the approach assumes that the impact of a Trojan on delay and power is large enough to be distinguished from modeling inaccuracies and from inherent variability in the chip fabrication process. Wei et al. [118] exploit this variability to evade detection by such defenses.

Golden reference approaches [19] depackage, delayer, and optically image (using a high resolution microscope) a subset of chips to establish that they are Trojan free, and use the delay and power profiles of these chips as "known good." However, such testing of large, complex ICs with billions of nanometer sized transistors is expensive, error prone, and "stretches analytical capabilities to the limits" [110]. Furthermore, this approach assumes that malicious modifications are optically observable in the first place, for instance, as transistors or wires added to or removed from the circuit. Meanwhile, Becker et al. [28] design optically undetectable Trojans that change only the doping concentration of a few transistors in the chip.

Waksman and Sethumadhavan [115] propose a *run-time* defense: power cycle the chip in the field (disabling timers), and give the chip encrypted inputs (disabling hard-wired cheat codes). However, computing on encrypted inputs could have high cost; also, an adversary can still use a randomly chosen input sequence, or a chip aging sensor [118].

Design-time techniques apply hardware obfuscation [46, 70, 92], normally used to protect intellectual property, to Trojan deterrence, the assumption being that if the foundry cannot infer the chip's function, it cannot interfere with that function. A separate line of research studies mechanisms to detect Trojans inserted by a malicious designer *pre-fab* [104, 116]; this is complementary to Zebra's focus on an untrusted foundry.

In the taxonomy, Zebra can be understood as run-time monitoring. Compared to the techniques above, Zebra's requirement of a correctly manufactured and integrated verifier (§9) may be more burdensome. However, Zebra also provides much stronger assurance: it defends against arbitrary misbehavior, with a formal and comprehensive soundness guarantee.

Verifiable outsourcing. The last several years have seen a flurry of work in built systems based on probabilistic proof protocols [24, 25, 29, 31–33, 41, 51, 55, 56, 58, 59, 61, 76, 88, 98–101, 106, 108, 112, 114] (see [117] for a survey). For our present purposes, these works are in two categories. The first category descends from GKR's [63] interactive proof (IP) protocol, and includes CMT [55] and Allspice [112], both of which Zebra builds on (§2.2). The second category includes *arguments*: interactive arguments with preprocessing [99–101] and *non-interactive* arguments with preprocessing [24, 32, 36, 61, 76, 88]. The latter are sometimes known as SNARKs, for succinct non-interactive arguments of knowledge.

The IP-based schemes offer information-theoretic security and, when applicable, have more efficient verifiers, with low (or no) precomputation costs. However, arguments work over *non*-deterministic arithmetic circuits and hence apply to a

broader set of computations: ANSI C programs, RAM, cloud applications, set computations, databases, etc. [24, 29, 31, 33, 41, 51, 56, 76, 114]. In addition, arguments have lower round complexity than the IP schemes, and SNARKs go further: they offer non-interactivity, zero knowledge, public verifiability, etc. The trade-off is that arguments require cryptographic assumptions (standard ones for interactive arguments, non-falsifiable [86] ones for SNARKs). We were inspired by this activity: as noted in Section 9, our initial objective was to accelerate the GGPR/Pinocchio SNARK in hardware.

To our knowledge, there are no prior hardware implementations of probabilistic proof protocols. An intriguing middle ground is the GPU implementation of CMT [108]. This work exploits some of the parallelism in CMT (as noted in Figure 2 and Section 3.2), and achieves speedups versus a CPU implementation. However, Zebra needs far greater speedups. This is because Zebra is working in a new (almost: see below) model, where the requirement is that the prover's overhead *also* be lower than the baseline.

One work, by Ben-Sasson et al. [30], has articulated the goal of considering both the prover's and verifier's costs, when analyzing performance relative to the native baseline. Their context is different: theirs is a theory paper, they work with classical PCPs, and they assume that $\mathcal{V}$ has access to a PCP. By contrast, Zebra explicitly targets an implementation, and requires precomputation and interaction. Nevertheless, this is inspiring work, and the combined overhead in their setup indeed drops below the native baseline asymptotically. However, the point at which this occurs—the "concrete-efficiency threshold" [30]—is problem sizes on the order of 2^{43}, which is larger than any of the aforementioned works can handle.

11 Summary and conclusion

This paper has defined the problem of verifiable ASICs; given a solution, Zebra, in the form of ASIC implementations of a probabilistic proof protocol; and modeled, measured, and accounted, to evaluate Zebra's applicability. Zebra is not perfect. On the other hand, the performance goals that we articulated for it were stringent: (1) beat native execution, considering not just the verifier's costs but also the prover's, and (2) instantiate efficient and physically realizable hardware. In fact, it wasn't initially clear that these goals could be met at all. Viewed in this more forgiving light, Zebra's performance advantage, however narrow and modest, is encouraging (at least to us).

There is much future work: relaxing Zebra's constraints, including the need for auxiliary inputs; handling broader classes of computations; evaluating alternative baselines such as trusted CPUs; investigating other applications; experimenting with FPGA implementations; and perhaps even taping out an artifact that definitively shows the feasibility of verifiable ASICs. Most importantly, a grand challenge for the area is to develop probabilistic proof machinery that handles all relevant computations, not only in principle but also when translated to implementation substrates as unaccommodating as ASICs.

Acknowledgments

We thank Justin Thaler for optimizing CMT [107]; Greg Shannon for an inspiring conversation; Andrew J. Blumberg, Dan Boneh, and Justin Thaler for thoughtful comments; and the anonymous reviewers, whose meticulous attention improved the draft and the work. The authors were supported by NSF grants CNS-1423249, CNS-1514422, CNS-1505728, CNS-0845811, TC-1111781, CNS-1527072, and SRC-2015-TS-2635; AFOSR grant FA9550-15-1-0302; ONR grant N00014-14-1-0469; a Microsoft Faculty Fellowship; and a Google Faculty Research Award.

Zebra's source code is available at:
`http://www.pepper-project.org/`

References

[1] `https://github.com/pepper-project`.
[2] AMD Radeon R9 270X. `https://www.techpowerup.com/gpudb/2466/radeon-r9-270x.html`.
[3] Cadence Suite. `http://www.cadence.com/`.
[4] Dell warns of hardware trojan. `http://http://homelandsecuritynewswire.com/dell-warns-hardware-trojan`.
[5] GeForce GTX 980. `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-980/specifications`.
[6] Icarus Verilog. `http://iverilog.icarus.com/`.
[7] IEEE standard 1800-2012: SystemVerilog. `https://standards.ieee.org/findstds/standard/1800-2012.html`.
[8] NanGate FreePDK45 open cell library. `http://www.nangate.com/?page_id=2325`.
[9] NCSU CDK. `http://www.cs.utah.edu/~elb/cadbook/Chapters/AppendixC/technology_files.html`.
[10] Protecting against gray market and counterfeit goods. `http://blogs.cisco.com/news/protecting_against_gray_market_and_counterfeit_goods`.
[11] Secure channel. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa380123(v=vs.85).aspx`.
[12] Semiconductors: markets and opportunities. `http://www.ibef.org/download/Semiconductors_220708.pdf`.
[13] Tezzaron Semiconductor. "Can you trust your chips?". `http://www.tezzaron.com/can-you-trust-your-chips`.
[14] Things that use Curve25519. `https://ianix.com/pub/curve25519-deployment.html`.
[15] Trusted foundry program. `http://www.dmea.osd.mil/trustedic.html`.
[16] Lagrange interpolation formula. In M. Hazewinkel, editor, *Encyclopedia of Mathematics*. Springer, 2001.
[17] libsnark. `https://github.com/scipr-lab/libsnark`, 2016.
[18] S. Adee. The hunt for the kill switch. `http://spectrum.ieee.org/semiconductors/design/the-hunt-for-the-kill-switch`, May 2008.
[19] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. In *IEEE S&P*, May 2007.
[20] Y. Arbitman, G. Dogon, V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFTX: A proposal for the SHA-3 standard. NIST submission, 2008.
[21] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Chapman & Hall/CRC, 2005.
[22] L. Babai. Trading group theory for randomness. In *STOC*, May 1985.
[23] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, May 1991.

[24] M. Backes, M. Barbosa, D. Fiore, and R. M. Reischuk. ADSNARK: Nearly practical and privacy-preserving proofs on authenticated data. In *IEEE S&P*, May 2015.

[25] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *ACM CCS*, Nov. 2013.

[26] M. Banga and M. S. Hsiao. A region based approach for the identification of hardware Trojans. In *HOST*, June 2008.

[27] C. Batten, A. Joshi, J. Orcutt, A. Khilo, B. Moss, C. Holzwarth, M. Popović, H. Li, H. Smith, J. Hoyt, F. Kärtner, R. Ram, V. Stojanović, and K. Asanović. Building manycore processor-to-DRAM networks with monolithic silicon photonics. In *IEEE HOTI*, Aug. 2008.

[28] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson. Stealthy dopant-level hardware trojans. In *CHES*, Aug. 2013.

[29] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza. Decentralized anonymous payments from Bitcoin. In *IEEE S&P*, May 2014.

[30] E. Ben-Sasson, A. Chiesa, D. Genkin, and E. Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. In *STOC*, June 2013.

[31] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO*, Aug. 2013.

[32] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO*, Aug. 2014.

[33] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security*, Aug. 2014.

[34] D. J. Bernstein. Curve25519: new Diffie-Hellman speed records. In *PKC*, Apr. 2006.

[35] S. Bhunia, M. Hsiao, M. Banga, and S. Narasimhan. Hardware Trojan attacks: threat analysis and countermeasures. *Proceedings of the IEEE*, 102(8):1229–1247, Aug. 2014.

[36] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *ITCS*, Jan. 2012.

[37] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer. Recursive composition and bootstrapping for SNARKs and proof-carrying data. In *STOC*, 2013.

[38] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic eurve cryptography (ECC) cipher suites for transport layer security (TLS), May 2006.

[39] G. Brassard, D. Chaum, and C. Crépeau. Minimum disclosure proofs of knowledge. *J. of Comp. and Sys. Sciences*, 37(2):156–189, Oct. 1988.

[40] B. Braun. Compiling computations to constraints for verified computation. UT Austin Honors thesis HR-12-10, Dec. 2012.

[41] B. Braun, A. J. Feldman, Z. Ren, S. Setty, A. J. Blumberg, and M. Walfish. Verifying computations with state. In *SOSP*, Nov. 2013.

[42] M. Bushnell and V. D. Agrawal. *Essentials of electronic testing for digital, memory and mixed-signal VLSI circuits*, volume 17. Springer Science & Business Media, 2000.

[43] D. Byrne, B. Kovak, and R. Michaels. Offshoring and price measurement in the semiconductor industry. In *Measurement Issues Arising from the Growth of Globalization*, Nov. 2009.

[44] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE TCAD*, 20(9):1059–1076, Sept. 2001.

[45] G. Carlson. Trusted foundry: the path to advanced SiGe technology. In *IEEE CSICS*, Nov. 2005.

[46] R. S. Chakraborty and S. Bhunia. HARPOON: an obfuscation-based SoC design methodology for hardware protection. *IEEE TCAD*, 28(10):1493–1502, Oct. 2009.

[47] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia. MERO: A statistical approach for hardware Trojan detection. In *CHES*, Sept. 2009.

[48] S. Checkoway, M. Fredrikson, R. Niederhagen, A. Everspaugh, M. Green, T. Lange, T. Ristenpart, D. J. Bernstein, J. Maskiewicz, and H. Shacham. On the practical exploitability of Dual EC in TLS implementations. In *USENIX Security*, Aug. 2014.

[49] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Intl. Conf. on Fault Tolerant Computing*, June 1978.

[50] A. Chiesa and E. Tromer. Proof-carrying data and hearsay arguments from signature cards. In *ICS*, Jan. 2010.

[51] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *EUROCRYPT*, Apr. 2015.

[52] C. Christensen, S. King, M. Verlinden, and W. Yang. The new economics of semiconductor manufacturing. `http://spectrum.ieee.org/semiconductors/design/the-new-economics-of-semiconductor-manufacturing`.

[53] E.-Y. Chung, C.-I. Son, K. Bang, D. Kim, S.-M. Shin, and S. Yoon. A high-performance solid-state disk with double-data-rate NAND flash memory. arXiv:1502.02239, 2009.

[54] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[55] G. Cormode, M. Mitzenmacher, and J. Thaler. Practical verified computation with streaming interactive proofs. In *ITCS*, Jan. 2012.

[56] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur. Geppetto: Versatile verifiable computation. In *IEEE S&P*, May 2015.

[57] F. Doany. Power-efficient, high-bandwidth optical interconnects for high-performance computing. `http://www.hoti.org/hoti20/slides/Fuad_Doany_IBM.pdf`, 2012. IEEE HOTI Keynote.

[58] D. Fiore, R. Gennaro, and V. Pastro. Efficiently verifiable computation on encrypted data. In *ACM CCS*, Nov. 2014.

[59] M. Fredrikson and B. Livshits. ZØ: An optimizing distributing zero-knowledge compiler. In *USENIX Security*, Aug. 2014.

[60] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *CRYPTO*, Aug. 2010.

[61] R. Gennaro, C. Gentry, B. Parno, and M. Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT*, 2013.

[62] C. Gentry and D. Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *STOC*, June 2011.

[63] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. *J. ACM*, 62(4):27:1–27:64, Aug. 2015. Prelim version STOC 2008.

[64] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. on Comp.*, 18(1):186–208, 1989.

[65] B. Grow, C.-C. Tschang, C. Edwards, and D. Burnsed. Dangerous fakes. *Business Week*, Oct. 2008.

[66] L. Henzen, P. Gendotti, P. Guillet, E. Pargaetzi, M. Zoller, and F. Gürkaynak. Developing a hardware evaluation method for SHA-3 candidates. In *CHES*. Aug. 2010.

[67] B. Hoefflinger. ITRS: The international technology roadmap for semiconductors. In *Chips 2020*. Springer, 2012.

[68] D. Hwang, K. Tiri, A. Hodjat, B. Lai, S. Yang, P. Shaumont, and I. Verbauwhede. AES-based security coprocessor IC in 0.18-μm CMOS with resistance to differential power analysis side-channel attacks. *IEEE JSSC*, 41(4):781–792, Apr. 2006.

[69] Top 13 foundries account for 91 percent of total foundry sales in 2013. `http://www.icinsights.com/news/bulletins/top-13-foundries-account-for-91-of-total-foundry-sales-in-2013/`, Jan. 2014.

[70] F. Imeson, A. Emtenan, S. Garg, and M. V. Tripunitara. Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation. In *USENIX Security*, Aug. 2013.

[71] M. Immonen. Development of optical interconnect PCBs for high-speed electronic systems—fabricator's view. `http://www-03.ibm.com/procurement/proweb.nsf/objectdocswebview/file2011+ibm+pcb+symposium+ttm/$file/ttm_optical+interconnect_ext.pdf`, 2011. IBM PCB Symposium 2011.

[72] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *IEEE CCC*, June 2007.

[73] Y. Jin and Y. Makris. Hardware Trojan detection using path delay fingerprint. In *HOST*, June 2008.

[74] J. Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *STOC*, May 1992.

[75] D. H. Kim, K. Athikulwongse, M. B. Healy, M. M. Hossain, M. Jung, I. Khorosh, G. Kumar, Y.-J. Lee, D. L. Lewis, T.-W. Lin, C. Liu, S. Panth, M. Pathak, M. Ren, G. Shen, T. Song, D. H. Woo, X. Zhao, J. Kim, H. Choi, G. H. Loh, H.-H. S. Lee, and S. K. Lim. Design and analysis of 3D-MAPS. *IEEE Trans. Computers*, 64(1):112–125, Jan. 2015.

[76] A. E. Kosba, D. Papadopoulos, C. Papamanthou, M. F. Sayed, E. Shi, and N. Triandopoulos. TRUESET: Faster verifiable set computations. In *USENIX Security*, Aug. 2014.

[77] F. Koushanfar and A. Mirhoseini. A unified framework for multimodal submodular integrated circuits trojan detection. *IEEE TIFS*, 6(1):162–174, Dec. 2011.

[78] J. Li and J. Lach. At-speed delay characterization for IC authentication and trojan horse detection. In *HOST*, June 2008.

[79] S. K. Lim. 3D-MAPS: 3D massively parallel processor with stacked memory. In *Design for High Perf., Low Power, and Reliable 3D Integrated Circuits*. Springer, 2013.

[80] C. Lund, L. Fortnow, H. J. Karloff, and N. Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 39(4):859–868, Oct. 1992.

[81] J. Markoff. Old trick threatens the newest weapons. *The New York Times*, Oct. 2009.

[82] S. Maynard. Trusted manufacturing of integrated circuits for the Department of Defense. In *National Defense Industrial Association Manufacturing Division Meeting*, 2010.

[83] S. Micali. Computationally sound proofs. *SIAM J. on Comp.*, 30(4):1253–1298, 2000.

[84] P. Milder, F. Franchetti, J. Hoe, and M. Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM TODAES*, 17(2):15:1–15:33, Apr. 2012.

[85] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Math. of Computation*, 48(177):243–264, Jan. 1987.

[86] M. Naor. On cryptographic assumptions and challenges. In *CRYPTO*, 2003.

[87] M. H. Nazari and A. Emami-Neyestanak. A 24-Gb/s double-sampling receiver for ultra-low-power optical communication. *IEEE JSSC*, 48(2):344–357, Feb. 2013.

[88] B. Parno, C. Gentry, J. Howell, and M. Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P*, May 2013.

[89] H. Pilo, J. Barwin, G. Braceras, C. Browning, S. Burns, J. Gabric, S. Lamphier, M. Miller, A. Roberts, and F. Towler. An SRAM design in 65nm and 45nm technology nodes featuring read and write-assist circuits to expand operating voltage. In *IEEE VLSI*, June 2006.

[90] E. P. Quevy. CMEMS technology: Leveraging high-volume CMOS manufacturing for MEMS-based frequency control. Technical report, Silicon Laboratories.

[91] J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic. *Digital integrated circuits*, volume 2. Prentice Hall Englewood Cliffs, 2002.

[92] J. A. Roy, F. Koushanfar, and I. L. Markov. EPIC: Ending piracy of integrated circuits. In *DATE*, Mar. 2008.

[93] M. Sako, Y. Watanabe, T. Nakajima, J. Sato, K. Muraoka, M. Fujiu, F. Kouno, M. Nakagawa, M. Masuda, K. Kato, Y. Terada, Y. Shimizu, M. Honma, A. Imamoto, T. Araya, H. Konno, T. Okanaga, T. Fujimura, X. Wang, M. Muramoto, M. Kamoshida, M. Kohno, Y. Suzuki, T. Hashiguchi, T. Kobayashi, M. Yamaoka, and R. Yamashita. A low-power 64Gb MLC NAND-flash memory in 15nm CMOS technology. In *IEEE ISSCC*, Feb. 2015.

[94] K. Sakuma, S. Skordas, J. Zitz, E. Perfecto, W. Guthrie, L. Guerin, R. Langlois, H. Liu, K. Ramachandran, W. Lin, K. Winstel, S. Kohara, K. Sueoka, M. Angyal, T. Graves-Abe, D. Berger, J. Knickerbocker, and S. Iyer. Bonding technologies for chip level and wafer level 3D integration. In *IEEE ECTC*, May 2014.

[95] P. Sasdrich and T. Güneysu. Efficient elliptic-curve cryptography using Curve25519 on reconfigurable devices. In *ARC*, Apr. 2014.

[96] P. Sasdrich and T. Güneysu. Implementing Curve25519 for side-channel–protected elliptic curve cryptography. *ACM TRETS*, 9(1):3:1–3:15, Nov. 2015.

[97] C. L. Schow, F. E. Doany, C. Chen, A. V. Rylyakov, C. W. Baks, D. M. Kuchta, R. A. John, and J. A. Kash. Low-power 16×10 Gb/s bi-directional single chip CMOS optical transceivers operating at < 5 mW/Gb/s/link. *IEEE JSSC*, 44(1):301–313, Jan. 2009.

[98] S. Setty, A. J. Blumberg, and M. Walfish. Toward practical and unconditional verification of remote computations. In *HotOS*, May 2011.

[99] S. Setty, B. Braun, V. Vu, A. J. Blumberg, B. Parno, and M. Walfish. Resolving the conflict between generality and plausibility in verified computation. In *EuroSys*, Apr. 2013.

[100] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making argument systems for outsourced computation practical (sometimes). In *NDSS*, Feb. 2012.

[101] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security*, Aug. 2012.

[102] A. Shamir. IP = PSPACE. *J. ACM*, 39(4):869–877, Oct. 1992.

[103] V. Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`.

[104] C. Sturton, M. Hicks, D. Wagner, and S. T. King. Defeating UCI: Building stealthy and malicious hardware. In *IEEE S&P*, May 2011.

[105] M. Tehranipoor and F. Koushanfar. A survey of hardware Trojan taxonomy and detection. *IEEE DT*, 27(1):10–25, Jan. 2010.

[106] J. Thaler. Time-optimal interactive proofs for circuit evaluation. In *CRYPTO*, Aug. 2013.

[107] J. Thaler. A note on the GKR protocol. http://people.seas.harvard.edu/ jthaler/GKRNote.pdf, 2015.

[108] J. Thaler, M. Roberts, M. Mitzenmacher, and H. Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud Workshop*, June 2012.

[109] K. Tiri, D. Hwang, A. Hodjat, B. Lai, S. Yang, P. Shaumont, and I. Verbauwhede. AES-based cryptographic and biometric security coprocessor IC in 0.18-μm CMOS resistant to side-channel power analysis attacks. In *VLSI Circuits*, June 2005.

[110] R. Torrance and D. James. The state-of-the-art in IC reverse engineering. In *CHES*, Sept. 2009.

[111] S. Trimberger. Trusted design in FPGAs. In *DAC*, June 2007.

[112] V. Vu, S. Setty, A. J. Blumberg, and M. Walfish. A hybrid architecture for interactive verifiable computation. In *IEEE S&P*, May 2013.

[113] R. S. Wahby, M. Howald, S. Garg, a. shelat, and M. Walfish. Verifiable ASICs. Cryptology ePrint Archive, Report 2015/1243, 2015.

[114] R. S. Wahby, S. Setty, Z. Ren, A. J. Blumberg, and M. Walfish. Efficient RAM and control flow in verifiable outsourced computation. In *NDSS*, Feb. 2015.

[115] A. Waksman and S. Sethumadhavan. Silencing hardware backdoors. In *IEEE S&P*, May 2011.

[116] A. Waksman, M. Suozzo, and S. Sethumadhavan. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *ACM CCS*, Nov. 2013.

[117] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, 58(2):74–84, Feb. 2015.

[118] S. Wei, K. Li, F. Koushanfar, and M. Potkonjak. Hardware Trojan horse benchmark via optimal creation and placement of malicious circuitry. In *DAC*, June 2012.

[119] S. Wei, S. Meguerdichian, and M. Potkonjak. Malicious circuitry detection using thermal conditioning. *IEEE TIFS*, 6(3):1136–1145, Sept. 2011.

[120] F. Wolff, C. Papachristou, S. Bhunia, and R. S. Chakraborty. Towards Trojan-free trusted ICs: Problem analysis and detection scheme. In *DATE*, Mar. 2008.

[121] Y. Zhou and D. Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. Cryptology ePrint Archive, Report 2005/388, 2005.