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 (an ASIC) 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 explores an alternative: using verifiable computation (VC), an untrusted ASIC computes *proofs* of correct execution, which are verified by a trusted processor or ASIC. Notably, in the present setting, the prover and verifier *together* must impose less overhead than the baseline alternative of running the given computation directly on the trusted platform. We respond to this challenge 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 interactive proof protocol; instantiating Zebra required a blend of new observations about CMT, careful hardware design, and attention to architectural challenges. We measure and evaluate Zebra; for a class of real computations, it indeed poses less overhead than executing directly on the trusted platform.

1 Introduction

This paper explores a new response to the threat posed by untrusted hardware fabrication. The threat exists 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. In such a case, the foundry can mount a *hardware Trojan* [14] attack by including malware inside the ASIC. Government agencies and semiconductor vendors have long regarded this threat as a core strategic concern [3, 8, 11, 13, 35, 66, 85].

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 [56] (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 [9], 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 [79] include post-fab testing on particular input patterns [37, 94], post-fab power and delay profiling [22, 59, 62, 63, 93], destructive delayering and imaging in combination with profiling [15], and power cycling in the field [89]. 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 the classic N-version technique [38]: use two foundries, deploy the two ASICs together, and, if their outputs differ, a trusted processor can take action (notify an operator, impose fail-safe behavior, etc.). This technique provides no assurance if the foundries collude—a distinct possibility, in light of the aforementioned characteristics of the semiconductor industry. A high assurance variant is to execute the desired functionality in software or hardware on a trusted platform (say, produced by a foundry in the same trust domain as the designer), 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 (weak) 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 [18, 49–51, 65, 77], efficient arguments [31, 58, 60, 68], SNARGs [48], SNARKs [30], and verifiable outsourced computation [19, 46, 49] 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 [20, 21, 24, 26, 28, 29, 32, 39, 42–45, 47, 61, 71, 74–76, 80, 82, 87, 88]. 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 at least $10^7\times$, greater than the cost of just executing the computation [91, Fig. 5].

[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 [12]. Furthermore, these costs worsen as the *technology node*—the length of the smallest transistor that can be fabricated, also referred to as the *critical dimension*—improves.

[2] This is different from, and complementary to, the vast literature on hardware verification, where the emphasis is on statically verifying that the intended circuit design (which is assumed to be manufactured faithfully) meets a higher-level specification.

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 [72]. As a consequence, the performance improvement, when going from an ASIC manufactured in a trusted (older) foundry to one manufactured in an advanced but untrusted foundry, can be *larger* than the overhead of provers in the aforementioned systems (as with the earlier example of India).

This gap implies that our motivating question potentially has a positive answer, using protocols for verifiable outsourcing. However, this picture can be brought to life only if the prover can be implemented on an ASIC in the first place—which, owing to the constraints of hardware, is easier said than done. Many protocols for verifiable outsourcing [20, 21, 24, 26, 28, 29, 32, 39, 43–45, 47, 61, 71, 74–76, 88] have concrete bottlenecks (cryptographic operations, serial phases, communication patterns that lack temporal and spatial locality, etc.) that seem inconsistent 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's algorithm uses no cryptographic operations (only field operations), has highly structured and parallel data flows, and demonstrates excellent spatial and temporal locality—all of which could plausibly lead to physical realizability. This is CMT [42] (an interactive proof that refines Muggles [49]). To be clear, CMT (with improvements [87]) applies principally to "quasi straight-line" computations (a term we clarify in Section 2.2). However, we can live with this restriction because there are computations that have the required form—particularly those that one would have thought to implement in an ASIC to begin with (§8).

Moreover, one might naturally expect something to be sacrificed (in this case, generality), 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 [32, 42, 46, 71, 74–76, 87, 88], 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, which is to design and implement a physically realizable, high-throughput ASIC for a prover and verifier based on CMT; address some of the accompanying architectural and operational challenges; and carefully analyze, model, and experiment to understand when the overall system, which we call *Zebra*, is truly motivated.

Zebra incorporates work at multiple levels of abstraction (§3). First, Zebra makes new observations about CMT's algorithms to extract additional parallelism (beyond that of prior work [82]) and then exploits those observations in its design. One level down, Zebra arranges for locality and predictable data flow: it avoids RAM where the algorithm seems to call for it, and avoids global communication and synchronization. At the lowest level, we give Zebra a latency insensitive design [34], with few timing constraints; and Zebra strategically reuses modules and circuitry to reduce ASIC area. In addition, Zebra responds to architectural challenges (§4), including how to meet the requirement, which exists in most implemented protocols for verifiable computation, of offline pre-processing work by the verifier; how to endow the verifier with storage without incurring the cost of a trusted storage substrate; and how to limit the costs of the verifier-prover communication.

The core design is fully implemented (§6): Zebra includes a compiler that takes an arithmetic circuit description directly to a synthesizable Verilog implementation. Combined with existing compilers that take C code to arithmetic circuit descriptions [26, 28, 29, 32, 39, 43, 71, 74, 76, 87, 88], Zebra obtains a pipeline in which a human writes high-level software, and a toolchain produces a hardware design.

Our evaluation of Zebra (§7) is based on detailed modeling (§5) and measurement. Taking into account energy, area, and throughput, Zebra outperforms the baseline when all of the following hold: the technology gap between $\mathscr{P}$ and $\mathscr{V}$ is at least seven generations; the computation of interest involves tens of thousands of operations; and these operations are "expensive" (i.e., a quarter or more are multiplies). A specific example is the number theoretic transform (§8): for 2^{10}-point transforms, Zebra is competitive with the baseline, and on larger computations it outperforms the baseline by 2–4×.

Though these results are encouraging, Zebra has clear limitations. Even under its applicability regime (noted earlier), it provides a gain versus the baseline only for relatively large computations. And even when it provides a gain, the price of verifiability is very high, when comparing to *untrusted* execution. Furthermore, it does not offer certain properties met by other systems for verifiable computation (low round complexity, $O(1)$ verifier complexity, public verifiability, zero knowledge properties, etc.); on the other hand, these amenities aren't needed in our context. Finally, Zebra's verifier requires periodic refreshing of pre-computed advice strings.

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

- It initiates the study of verifiable ASICs, and demonstrates their feasibility as a response to hardware Trojans, one that works in a much stronger threat model than most prior work on hardware Trojans. The high-level picture was folklore, but there have been many details to work through.
- It makes new observations about, and refinements to, 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 none is worthy of its own paper, together they produce a milestone: the first hardware design and implementation of a probabilistic proof system.
- It performs careful modeling, accounting, and measurement to characterize performance. At a higher level, this is the first

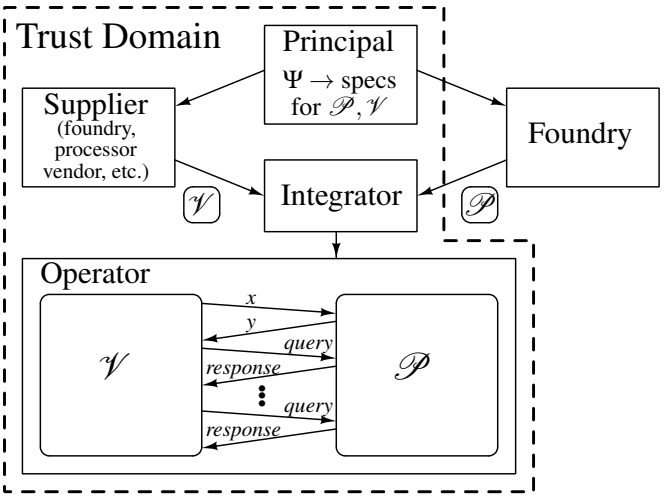


FIGURE 1—Verifiable ASICs. A principal outsources the production of an ASIC ($\mathscr{P}$) to an untrusted foundry and gains high-assurance execution via a trusted verifier ($\mathscr{V}$) and a probabilistic proof protocol.

work to identify a setting in which one can simultaneously capture the "cost" of the prover and verifier together, and to give an implementation of the prover and verifier for which this quantity is less expensive than having the verifier compute on its own.

2 Problem, background, and approach

2.1 Problem statement: verifiable ASICs

Setup and threat model. 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* $\mathscr{P}$, is assumed to remain within the trust domain. In particular, there is a trusted step in which an *integrator* produces a single system by combining $\mathscr{P}$ with a trusted component, called a *verifier* $\mathscr{V}$. Furthermore, the operator (or end-user) trusts the system that the principal delivers to it.

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

$\mathscr{P}$ 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: $\mathscr{P}$ could have been designed maliciously, manufactured with an arbitrariliy modified design [79], replaced with a counterfeit [8, 52] en route to the principal's trust domain, and so on.

The technology node on which $\mathscr{V}$ executes is trusted by the principal. For example, $\mathscr{V}$ could be an ASIC manufactured at a less advanced foundry located onshore. Or $\mathscr{V}$ 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.

$\mathscr{V}$ is assumed to have a source of (pseudo)random bits. $\mathscr{V}$ also has private storage, configured by the principal, which can hold the result of any offline pre-computation required or permitted by the protocol.

There are, in this setup, requirements that surround the integration step. Our work does not specifically address them, so we regard them as assumptions for now. First, $\mathscr{P}$ should do *nothing more* than computing Ψ (and proving); in practice, this means $\mathscr{P}$ must communicate only with $\mathscr{V}$ and that $\mathscr{V}$'s internal state is inaccessible to $\mathscr{P}$. (This might necessitate separate power supplies, shielding $\mathscr{V}$ from electro-magnetic attacks [96], etc.; however, we acknowledge that completely eliminating covert channels is its own topic.) Second, $\mathscr{V}$ may need to include a fail-safe (such as a kill switch), to handle the case that $\mathscr{P}$ conducts a denial of service attack (by returning wrong answers, or by refusing to engage in the protocol). Finally, $\mathscr{V}$ may need to be electrically protected, to guard against a $\mathscr{P}$ that tries to disable it.

Performance goals. A solution to the above setup makes sense when—for the given technology nodes and the given computation Ψ—the cost of $\mathscr{V}$ and $\mathscr{P}$ together is less than the **native baseline** (§1): a chip (or CPU), in the same technology node as $\mathscr{V}$, that executes Ψ directly.

How should one measure costs? With hardware, choosing a metric is a delicate task because of trade-offs. As one example, a common concern is *area* (the size of the ASIC in square millimeters), yet area alone cannot be the metric: a design can, for example, iteratively re-use modules to lower area at the cost of lower throughput. One also cares about energy consumption because it drives the chip's operational cost, an important factor in many domains (datacenters, mobile computing, etc.).

The metric that we use is $E \cdot A_s/T$ (which we sometimes call **EAsT**): energy consumed by computing Ψ and running the proof protocol (E), times area consumed by the integrated system (A_s), divided by throughput (T). This metric (or a variant that captures delay, as opposed to throughput) is commonly used in hardware design [17, 40, 95].[3] A lower score is better. The term A_s is a weighted sum of area consumed in both the trusted and untrusted technology nodes; the untrusted area is divided by s. We leave s as a parameter, first, because s will vary with the principal; and second, because evaluating cost of silicon across technology nodes is a thorny topic, and arguably a research question [33, 67].

One thing that is not explicitly captured by the EAsT metric is *physical realizability*: one needs to be able to manufacture $\mathscr{P}$ (and $\mathscr{V}$, if it's an ASIC). In practice, this requirement will show up by constraining area (large chips have low manufacturing yield owing to defects, routing issues, etc.); it will also constrain

[3] Energy-delay (EDP) product is commonly used as a performance metric when evaluating software running on a CPU. In that case, silicon area is fixed, and for single threaded execution, $D = 1/T$.

power dissipation (because of heat), and thus the product of E and T. (Recall that E has units of joules/op and T of ops/second; $E \times T$ is in joules/second, i.e., watts.)

In addition, the metric captures only online costs. As noted above, the protocol may involve precomputation and hence offline costs. We will treat these costs later (§4).

2.2 Interactive proofs for verifiable computation

In responding to the requirements above, our starting point is a protocol that we call *OptimizedCMT*. Using an observation of Thaler [81] and additional simplifications and optimizations, this protocol—which we are *not* claiming as a contribution of this paper—optimizes CMT-slim [87, §3], which refines CMT [42, §3; 82], which refines GKR [49; 50, §3]; these are all interactive proofs [18, 49–51, 65, 77]. In the rest of this section, we describe OptimizedCMT; this description owes some textual debts to Vu et al. [87].

A verifier $\mathcal{V}$ and a prover $\mathcal{P}$ agree, offline, on a computation Ψ, which is expressed as an *arithmetic circuit (AC)* $\mathcal{C}$. 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 are assuming, for simplicity, that all layers, except for the input and output, have the same number of gates).

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

- **Completeness.** If $y = \mathcal{C}(x)$ and if $\mathcal{P}$ follows the protocol, then $\Pr\{\mathcal{V} \text{ accepts}\} = 1$.
- **Soundness.** If $y \neq \mathcal{C}(x)$, then $\Pr\{\mathcal{V} \text{ accepts}\} < \varepsilon$, 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.
- **Efficient verifier.** The verifier's *online* work can be less than computing Ψ; it is proportional to the length of the input and output, and the depth of Ψ times the logarithm of its width: $O(d \cdot \log G + |x| + |y|)$. This assumes $\mathcal{V}$ has access to advice strings that have been pre-computed offline and independently of the input, in time $O(d \cdot G \cdot \log G)$.
- **Efficient prover.** The prover's total work is polylogarithmically more expensive than evaluating the circuit itself, i.e., $O(d \cdot G \cdot \log^2 G)$.

Applicability. In theory, the protocol can be applied to any sufficiently "wide" computation and achieve work-savings for $\mathcal{V}$ (excluding the cost of pre-processing). But in practice, it excels for computations that have a sub-linear number of branches (comparison operations and inequality checks), and do not rely on indirect memory addressing. (See Allspice [87, §4.3] for a more precise discussion of applicability.) One can think of OptimizedCMT as best suited to computations that have quasi-straightline arithmetic circuit implementations; we note that several computational tasks of interest follow this form (§8).

Protocol details. Within a layer of the AC, gates are numbered between 1 and G and have a *label* corresponding to the binary representation of their number, viewed as an element of $\mathbb{F}_2^b$ 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, and its outputs (y) are viewed as being at layer 0. At each layer $i = 0, \ldots, d$, the *evaluator function* $V_i \colon \mathbb{F}_2^b \to \mathbb{F}$ 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 and that $V_0(j)$ returns the j^{th} output element.

Observe that $\mathcal{C}(x) = y$, meaning that y is the correct output, if and only if $V_0(j) = y_j$, for all output gates j. However, $\mathcal{V}$ cannot check directly whether this condition holds: evaluating $V_0(\cdot)$ would require re-executing the circuit (which is ruled out by the problem statement). Instead, the protocol allows $\mathcal{V}$ to efficiently reduce a condition on $V_0(\cdot)$ to a condition on $V_1(\cdot)$. That condition also cannot be checked (because it would require executing most of the circuit), but the process can be iterated until it produces a condition that $\mathcal{V}$ 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 a *wiring predicate* $\text{add}_i \colon \mathbb{F}_2^{3b} \to \mathbb{F}$, 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 $g \colon \mathbb{F}_2^m \to \mathbb{F}$, 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 will notate multilinear extensions with tildes. Thaler [81], building on GKR [49, 50], shows the following:

$$\tilde{V}_{i-1}(q) = \sum_{z_0,z_1 \in \mathbb{F}_2^b} \big(\tilde{\text{add}}_i(q, z_0, z_1) \cdot \big(\tilde{V}_i(z_0) + \tilde{V}_i(z_1)\big) + \tilde{\text{mult}}_i(q, z_0, z_1) \cdot \tilde{V}_i(z_0) \cdot \tilde{V}_i(z_1)\big) \quad (1)$$

The signatures are $\tilde{V}_i, \tilde{V}_{i-1} \colon \mathbb{F}^b \to \mathbb{F}$ and $\tilde{\text{add}}_i, \tilde{\text{mult}}_i \colon \mathbb{F}^{3b} \to \mathbb{F}$.

At this point, $\tilde{V}_{i-1}(\cdot)$ is in a form that calls for a *sum-check protocol* [65] (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).

Figure 2 depicts pseudocode for $\mathcal{P}$ and $\mathcal{V}$. We do not have space to justify many of these details. (See [50, §3][42, §A.1–A.2][87, §2.2,§A][81].) Instead, we want to communicate the

```
1: function PROVE(Circuit c, input x)
2:     q_0 ← ReceiveFromVerifier()    // see line 56
3:     d ← c.depth
4:
5:     // each circuit layer induces one sum-check 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:
12: function SUMCHECKP(Circuit c, layer i, q_{i-1})
13:     for j = 1,...,2b do
14:
15:         // compute F_j(0), F_j(1), F_j(2)
16:         parallel for all gates g at layer i − 1 do
17:             for k = 0,1,2 do
18:                 // below, s ∈ F_2^{3b}. s is a gate triple in binary.
19:                 s ← (g, g_L, g_R) // g_L, g_R are labels of g's layer-i inputs
20:
21:                 u_k ← (q_{i-1}[1],...,q_{i-1}[b], r[1],...,r[j−1], k)
22:                 // notation: χ : F → F. χ_1(t) = t, χ_0(t) = 1−t
23:                 termP ← ∏_{ℓ=1}^{b+j} χ_{s[ℓ]}(u_k[ℓ])
24:
25:                 if j ≤ b then
26:                     termL ← Ṽ_i(r[1],...,r[j−1], k, g_L[j+1],...,g_L[b])
27:                     termR ← V_i(g_R) // V_i = Ṽ_i on gate labels
28:                 else    // b < j ≤ 2b
29:                     termL ← Ṽ_i(r[1],...,r[b])
30:                     termR ← Ṽ_i(r[b+1],...,r[j−1], k, g_R[j−b+1],...,g_R[b])
31:
32:                 if g is an add gate then
33:                     F[g][k] ← (termL + termR) · termP
34:                 else if g is a mult gate then
35:                     F[g][k] ← termL · termR · termP
36:
37:         for k = 0,1,2 do
38:             F_j[k] ← ∑_{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(Circuit 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:         // reduce h_0 ?= Ṽ_i(w_0), h_1 ?= Ṽ_i(w_1) to a_i ?= Ṽ_i(q_i):
64:         // • let H(t) = Ṽ_i((w_1 − w_0)t + w_0)
65:         // • we want H(0),...,H(b)
66:         // • h_0, h_1 should be H(0), H(1); now expect 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 [42, 49, 50, 81, 87], expressed in terms of the parallelism used by TRMP [82]. Some of the notation and framing is borrowed from [87]. 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 [50, §3][42, §A.1–A.2][87, §2.2,§A][81] for explanation of all details.

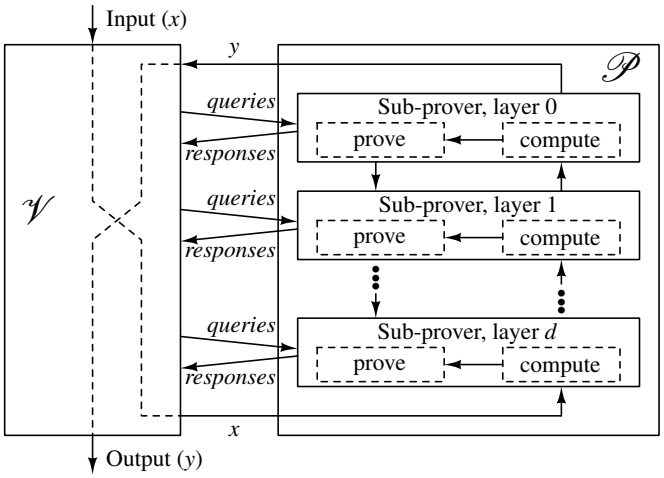


FIGURE 3—Prover's architecture. Each logical sub-prover handles all of the work for a layer in a layered arithmetic circuit $\mathscr{C}$ of width G and depth d.

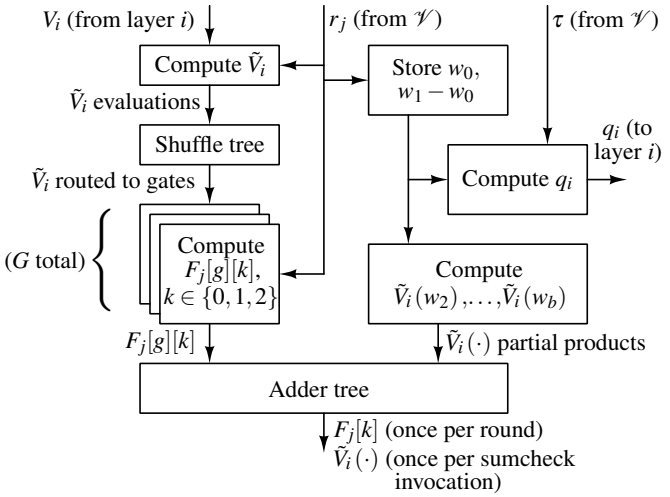


FIGURE 4—Design of a sub-prover.

structure of the protocol, as well as what work $\mathscr{P}$ must perform. Specifically, there is one *invocation* of the sum-check protocol for each of the $i = 1,\ldots,d$ layers of the circuit. Within such an invocation, there are $2b$ *rounds* (logarithmic in the number of gates at a layer). Finally, notice that—despite Equation (1)—$\mathscr{P}$ does not explicitly sum over a hypercube. The depicted sum, in which each gate makes a constant contribution to each round of the sum-check protocol, is established in CMT [42, §A.2].

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. As noted in the introduction, Zebra begins from the observation that OptimizedCMT seems amenable to implementation in hardware. To go from this observation to a physically realizable, high-throughput, area-efficient design, Zebra exploits new and existing observations about OptimizedCMT, and uses these observations at multiple levels of abstraction. Zebra's specific design ethos (for both the prover and verifier) is as follows:

- *Extract parallelism.* This does not reduce the total work that must be done, but it does lead to speedups. Specifically, holding area constant, more parallelism yields better throughput.
- *Exploit locality.* This means avoiding unnecessary communication among individual modules: communication imposes constraints during the synthesis process,[4] which interferes with physical realizability. Locality also refers to avoiding dependencies among modules; in hardware, dependencies translate to timing relationships, timing relationships create serialization points, and serialization harms throughput.
- *Reuse work.* This means both reusing computation results (saving energy), and reusing the same modules for different parts of the computation (saving area). However, this reuse must be carefully engineered to avoid interfering with the previous two goals.

[4] *Synthesis* is the process of translating a high-level description of a digital circuit design into a concrete implementation in primitive logic gates. This translation must satisfy a set of *timing constraints*, i.e., relationships between signals in the design. Such constraints can be explicit (e.g., the designer specifies the frequency of the master clock), or they can be implicit (e.g., if one flip-flop's input relies on a combinational function of another flip-flop's output, this imposes a constraint on the timing relationship between the two).

3.1 Overview

Figure 3 depicts the top-level design of the prover. The prover comprises logically separate *sub-provers*, each multiplexed over one or more physical 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 will be used in the corresponding proof step. Zebra is agnostic about the number of *physical* sub-prover modules; the choice is a classical area-throughput trade-off (§3.2).

The design of a sub-prover is depicted in Figure 4. A sub-prover proceeds through sum-check 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 16), 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 leverages careful observations to reuse computation results, thus avoiding duplicate work (§3.4).

The design of the verifier is similar to that of the prover; however, the verifier's work is naturally more serial than the prover's, so its design aims to reuse modules (and thus save area) without introducing execution 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 [82]. Below, we describe further parallelism extracted by Zebra's design.

Exploiting layered circuits. The layering in the arithmetic circuit that OptimizedCMT works over (§2.2) creates a natural area-throughput trade-off. At one extreme, Zebra can conserve area, by having only a single physical sub-prover, which is iteratively "reused" for each layer of the circuit. 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 circuit, and arrange them in a classical pipeline. Specifically, the sub-prover for layer i handles successive *executions*, 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 executions; the acceleration in throughput, compared to using a single sub-prover for all layers, 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 all intermediate points (two logical sub-provers per physical prover, etc.). The fundamental source of the 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, each sub-prover must buffer results of executing its layer of $\mathscr{C}$ until it has executed the corresponding proof step. This results in a storage requirement proportional to the maximum number of computations "in-flight" at one time, independent of the number of physical sub-provers.

Gate-level parallelism. Within a sub-prover, and within a sum-check round, there is parallel proving work not only for each gate (as in prior work [82]) but also for each (gate, k) pair, for $k = \{0,1,2\}$. That is, in Zebra, the loops in lines 16 and 17 (Figure 2) are combined into a "parallel for all (g,k)". This is feasible in Zebra 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 23) 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 given gate and its gate prover; 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 sum-check 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 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 sum-check invocation, the prover apparently has a bottleneck: computing $\tilde{V}_i(w_2),\ldots,\tilde{V}_i(w_b)$ (line 50 in Figure 2) is costly and serialized. (Indeed, prior work [82] performs this step at the end.) However, Zebra observes that there is a way to compute these quantities in parallel with the rest of the invocation, using incremental computation and local state.

Computing $\tilde{V}_i(w_2),\ldots,\tilde{V}_i(w_b)$ can be performed with $(b-1)\cdot G\cdot b$ products ($= O(G\cdot\log^2 G)$).[5] This is because $\tilde{V}_i$ has to be evaluated at each of the $b-1$ vectors, and $\tilde{V}_i$ has the form:[6]

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

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. At first glance, the required evaluation work seems as though it 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 increasingly revealed over the remaining rounds. It is possible to exploit this observation.

Specifically, after round $b+1$, the prover can perform $(b-1)\cdot G$ of the required products. To see this, notice that 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 and retain $V_i(g)\cdot\prod_{\ell=1}^{2}\chi_{g[\ell]}(w_t[\ell])$, again for all $g \in \{1,\ldots,G\}$, $t \in \{2,\ldots,b\}$. 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 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 an adder tree.

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

[5] In fact, leveraging common factors among the partial products of each $\tilde{V}_i(\cdot)$ evaluation, it is possible to reduce this work to $O(G\cdot\log G)$; but doing so entails substantial communication and synchronization overhead. This is an example of the tension between reusing work and exploiting locality (§3). Future work is a more careful study of how to incorporate this optimization.

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

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: the P_g values (which are stored in accumulators within the functional blocks of the gate provers that need them), and 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 26 through 30, Figure 2), by evaluating $\tilde{V}_i(\cdot)$ at various points. Prior work [42, 82, 87] 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 $j \leq b$; for $j > b$, the picture is similar but not identical.) 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 $k = 0, 1$.[7] This requires updating (and shrinking) the lookup table at the end of each round, a step that can be performed efficiently because for all $T_{j+1} = (t_{j+1}, \ldots, t_b) \in \mathbb{F}_2^{b-j}$,

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

Zebra must implement equivalent logic (on-demand "lookup", and incremental update) but without using 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 (because, 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 sum-check 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). Despite the fact that gate provers complete their work at different times (owing to differences in, for example, mult and add), no additional control is required.

[7] 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).

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 (in part because of the precomputed advice strings; see §4 and line 100 of Figure 2).

To address this issue, Zebra makes two observations. First, the work of the first and last layers can happen in parallel; this work determines the length of a pipeline stage. Within that length, interior layers can be handled sequentially (for example, two or more interior layers can happen during one of the stages; 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 (through reuse), and in this case the savings do not detract from throughput. Together, these observations permit flexibility: the designer can choose parameters to optimize the ratio of the verifier's area and throughput (§5, §7.4).

4 System architecture

Having described the design of $\mathscr{P}$ and $\mathscr{V}$, we articulate several challenges of system architecture and operation, and walk through Zebra's responses.

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 [20, 21, 24, 26, 28, 29, 32, 39, 42–45, 47, 61, 71, 74–76, 80, 82, 87, 88]. They must therefore—if the goal is to save the verifier work—amortize the precomputation in one way

or another. In Zebra, the integrator is presumed to install the results of this precomputation as advice strings (§2.2) in $\mathcal{V}$'s private storage (§2.1); $\mathcal{V}$ consumes one advice string for each execution. This description raises three questions, which we answer below: (1) What is the required precomputation and advice string? (2) What happens when $\mathcal{V}$'s storage is exhausted? (3) How does the work of creating the advice strings amortize?

The required precomputation (whose asymptotic cost is given in Section 2.2) is to evaluate $\tilde{\text{add}}_i(q_{i-1}, r[1], \ldots, r[2b])$ and likewise for $\tilde{\text{mult}}_i$ (line 100, Figure 2), for $i = \{1, \ldots, d\}$ (as well as some lower order work to compute Lagrange coefficients, to accelerate the computation of H^* in line 70). As an important optimization, $\mathcal{V}$'s advice string includes, in addition to the aforementioned quantities, a pseudorandom *seed*. Using this seed, $\mathcal{V}$ can *rederive* the pseudorandom inputs to the precomputation (namely the $d \cdot (2b+1) + b - 1$ pseudorandom field elements, coming from lines 54, 68, and 78, in Figure 2). Without this optimization, $\mathcal{V}$ would need to store these quantities directly, because the protocol requires $\mathcal{V}$ to successively reveal them to $\mathcal{P}$. The optimization reduces $\mathcal{V}$'s storage costs by approximately 2/3. Per-execution, the advice string is the aforementioned evaluations, plus the seed, for a total of $d \cdot (b+2) + 1$ field elements.

When storage is exhausted, the principal must securely refresh with new advice strings computed offline. Thus, it is the responsibility of the principal to select the storage parameter according to the application. For example, at a throughput of 10^4 executions per second, and roughly 10^4 bytes per advice string, a 1TB memory suffices for 10^4 seconds.

To amortize this work, Zebra presumes that the integrator *reuses the precomputations over all* $\mathcal{V}$ (for example, over all $\mathcal{V}$ chips that are currently operating).[8] Preserving soundness in this regime requires that each extant prover is isolated, a stipulation of the setup (§2.1). Asymptotically, since pre-computing one advice string requires $O(d \cdot G \cdot \log G)$ offline computation, as long as the number of operating verifiers is substantially larger than $\log G$, then the overhead of the pre-computation can be amortized to negligible across the entire deployment of $\mathcal{V}$.

Managing storage costs. Given the number of precomputations that $\mathcal{V}$'s private storage must hold, it could impose an enormous area cost to build that storage in the trusted technology node. Zebra's solution is to endow $\mathcal{V}$ with *untrusted* storage, and to layer an authentication-encryption protocol. Specifically, $\mathcal{V}$ and the integrator are assumed to share an encryption key (that $\mathcal{V}$ can protect in a small amount of trusted storage). When $\mathcal{V}$'s integrator installs into (untrusted) storage, it provides the data in (authenticated) encrypted form using the key; when $\mathcal{V}$ retrieves, it decrypts, implicitly checking authentication and rejecting (implying halting, entering the fail-safe state, etc. (§2.1)) if tampering is detected. (Note that we do not need ORAM: given the protocol, $\mathcal{V}$'s access pattern is known.) We account for $\mathcal{V}$'s cost for AES in our modeling and evaluation (§5, §7).[9]

Communication and integration. The default integration approach—a printed circuit board—would limit bandwidth and impose high cost in energy (and thus EAsT) for communication. This would be problematic for Zebra because the protocol contains a lot of inter-chip communication (for $\mathcal{P}$, it is lines 8, 40, 41, 47, and 50; for $\mathcal{V}$, lines 67, 71, 82, 87, and 99). Moreover, the preceding paragraph added communication because $\mathcal{V}$'s storage is on a different chip. Zebra's response is to draw on *3D packaging technology* [64], in which ASICs can be vertically stacked and connected using a dense array of short vertical interconnects, referred to as through-silicon vias (TSVs). This enables high bandwidth and low energy communication between chips (§5). This arrangement requires the integrator (§2.1) to have access to 3D packaging technology. This assumption seems reasonable, first, because vertical interconnects, even in the most advanced 3D packages, are several times larger than the length of a transistor in a relatively mature technology. Moreover, several small vendors already provide trusted 3D packaging services to government agencies [10] (albeit for a different purpose [57]).

5 Cost analysis and accounting

This section presents an analytical model for the energy, area, and throughput costs of Zebra when applied to ACs of width G and depth d. The cost model serves as an expository tool to understand Zebra's performance on the EAsT metric. The following rough analogies hold: E is the number of operations in a single execution of OptimizedCMT; A_s is the parallelism with which Zebra executes; and T is limited by the critical path of execution.

Figure 5 summarizes the cost model. It covers the following operations: (1) protocol execution (*compute*; §2.2, Fig. 2); (2) communication (*tx*; §2.2); (3) storage (and decryption for $\mathcal{V}$; *store*; §4); and (4) for $\mathcal{V}$, pseudorandom number generation (*PRNG*; §4). In Section 7.1, estimates for the parameters are derived through synthesis and simulation.

Energy and area costs. The energy costs relate to the runtime invocations of operations (1)–(4) in $\mathcal{P}$ and $\mathcal{V}$. Area costs reflect Zebra's design, i.e., the number of hardware modules allocated, and the way those modules are shared among operations. We observe, and later affirm in Section 7.2, that field arithmetic dominates energy and area costs for $\mathcal{P}$ and $\mathcal{V}$. 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 slowest pipeline delay in either $\mathcal{V}$ or $\mathcal{P}$; that is, the pipeline delay of the faster component can be increased without hurting throughput. The cost model includes parame-

[8] Allspice, for example, handles the same issue with *batch verification* [87, §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.

[9] We find that in practice this cost is modest: the energy to decrypt one field element is about one-fifth the cost of a field multiply (§7.1, Figs. 6, 7). Meanwhile, $\mathcal{V}$ executes 7× more multiplications than decryptions (§5).

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

$n_{\mathscr{V},\text{io}}$: $\mathscr{V}$ parameter; trades area vs i/o delay — $\left\langle E_{\text{g,u}}\right\rangle$: mean per-gate energy of $\mathscr{C}$, untrusted — d: depth of arithmetic circuit $\mathscr{C}$
$n_{\mathscr{V},\text{sc}}$: $\mathscr{V}$ parameter; trades area vs sumcheck delay — $n_{\mathscr{P},\text{sc}}$: $\mathscr{P}$ parameter; trades area vs delay — G: number of gates in one layer of $\mathscr{C}$
$E_{\{\text{add,mul,tx,sto,prng}\},\{\text{t,u}\}}$: energy cost in {trusted, untrusted} technology node for {+, ×, transmit, store, PRNG}
$A_{\{\text{add,mul,tx,sto,prng}\},\{\text{t,u}\}}$: area cost in {trusted, untrusted} technology node for {+, ×, transmit, store, PRNG}
$\lambda_{\{\text{add,mul}\},\{\text{t,u}\}}$: delay in {trusted, untrusted} technology node for {+, ×}

FIGURE 5—$\mathscr{V}$ and $\mathscr{P}$ costs as a function of $\mathscr{C}$ parameters and technology nodes (simplified model). Energy and area of transmit, store, and PRNG indicate costs for a single element of $\mathbb{F}_p$. Transmit, store, and PRNG occur in parallel with execution; thus, 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.

ters to allow Zebra to trade increased pipeline delay for reduced area by removing functional units (e.g., sub-provers) in either $\mathscr{P}$ or $\mathscr{V}$ (§3.2, §3.5). Such trade-offs are typical in hardware design [69]; in the current context, they are used to optimize Zebra's EAsT score by balancing $\mathscr{V}$'s and $\mathscr{P}$'s delays (§7.4).

For $\mathscr{P}$, this parameter is $n_{\mathscr{P},\text{sc}}$: the number of physical sub-prover modules (§3.2). For $\mathscr{V}$, these parameters are $n_{\mathscr{V},\text{sc}}$ and $n_{\mathscr{V},\text{io}}$: roughly, the area apportioned to sum-check work versus computation over $\mathscr{C}$'s inputs and outputs, respectively (§3.5). During our evaluation and synthesis, these parameters will be constrained to keep area at or below some fixed size, to reflect manufacturability constraints.

Storage versus precomputation. Our cost model captures online costs, only. Thus, it accounts for the area and energy necessary to store, retrieve, and decrypt the precomputed advice strings (§4) but does not cover the costs of computing those strings (these costs were briefly treated in Section 4).

6 Implementation of Zebra

Our implementation of Zebra comprises four components.

The first component of Zebra is a compiler toolchain that produces $\mathscr{P}$. The toolchain takes as input a high-level description of an AC (in the format required for use with the Allspice compiler [1]), the implementation trade-off parameter $n_{\mathscr{P},\text{sc}}$ (§5, §7.4), 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 [5]).

Second, Zebra contains a parameterized implementation of $\mathscr{V}$ in SystemVerilog. As with $\mathscr{P}$, the designer supplies primitives for field arithmetic blocks in $\mathbb{F}_p$. Additionally, the designer selects the parameters $n_{\mathscr{V},\text{sc}}$ and $n_{\mathscr{V},\text{io}}$ (§5, §7.4).

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

Finally, Zebra implements a framework for cycle-accurate RTL simulations of complete interactions between $\mathscr{V}$ and $\mathscr{P}$. For this purpose, Zebra extends standard RTL simulators (tested with Cadence Incisive [2] and Icarus Verilog [4]) with an interface that abstracts communication between $\mathscr{P}$ and $\mathscr{V}$ (§2.2), and between $\mathscr{V}$ and its private memory (§4). The interface is written in C using the Verilog Procedural Interface (VPI) [5].

In total, our implementation comprises approximately 6000 lines of SystemVerilog, 9500 lines of C/C++ (partially inherited from Allspice [1]), 600 lines of Perl, and 300 lines of miscellaneous scripting glue.

7 Evaluation

In this section, we seek to answer: *For which ACs can Zebra outperform other techniques for high-assurance execution?*

We evaluate Zebra on synthetic benchmarks across a range of arithmetic circuit sizes and physical parameters. We find that, on computations to which Zebra applies—i.e., quasi-straightline computations (§2.2)—it can meet or exceed the performance of optimized baseline implementations when both (1) the technology gap between $\mathscr{V}$ and $\mathscr{P}$ is at least seven generations; and (2) the computation is "hard" for the baseline, i.e., it involves tens of thousands of operations, and those operations are expensive on average (that is, at least a quarter of them are multiplies, which are substantially costlier than adds; §7.1). At a high level, the first condition helps offset $\mathscr{P}$'s proving costs, and the second ensures that $\mathscr{V}$'s savings overwhelm the overhead of participating in the protocol (§2.2).

Section 8 makes this evaluation concrete by comparing Zebra to a real-world baseline on an application of real interest.

	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/($\mathbb{F}_p$/ns)
tx	1100	3400	600	1900
store	42×10^3	380×10^6	550	1900
PRNG	8700	17×10^6	—	—

FIGURE 7—Costs for communication, storage (including decryption, for $\mathscr{V}$), and PRNG. These costs are extrapolated from published results [53, 55, 64, 83, 86] using standard scaling models [54].

	$\log G$	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.

Method. Zebra's implementation allows us to evaluate using cycle-accurate SystemVerilog simulations, and by synthesizing directly to hardware. However, this approach comes with a practical limitation: synthesizing and simulating even a moderately-sized chip design can take hundreds of core-hours; meanwhile the aim here is to characterize Zebra over a wide range of ACs. Thus, we leverage the analytical cost model for Zebra described in Section 5. We do so in three steps.

First, we obtain values for the parameters of this model, i.e., the energy, area, and delay of Zebra's basic operations (§5, (1)–(4)), using a combination of hardware synthesis results and data published in the literature. This is described in Section 7.1.

Next, in Section 7.2, we validate the cost model by comparing predictions from the model with synthesis results of one full layer of $\mathscr{P}$, and cycle-accurate simulations of an entire Zebra instance. The synthesis and simulation data closely match our analytical predictions.

Finally, we use the validated cost model to compare Zebra against several baselines. A natural baseline for comparison is high-assurance execution in a trusted technology; in Section 7.3, we describe how we estimate the baseline costs, and in Section 7.4, we evaluate how Zebra relates to (and can outperform) the baseline. In Section 7.5, we compare Zebra with alternate approaches to high-assurance execution, i.e., lazy re-execution and execution on a trusted CPU. In Section 7.6, we briefly discuss the "price of verifiability" by comparing Zebra to an approach that simply foregoes high-assurance.

7.1 Estimating parameters

Synthesis of field operations. Figure 6 reports synthesis data for both field operations in $\mathbb{F}_p$ for prime $p = 2^{61} - 1$. This prime allows for an efficient and easily implementable modular reduction. Both operations were written in Verilog and synthesized using Cadence RTL Compiler to two technology libraries: Nangate 45 nm Open Cell [6] and a 350 nm library from NC State University and the University of Utah [7].

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 take results from CPUs built with 3D packaging [64]; longitudinal studies of RAM costs [86]; and ASICs for cryptographic operations [53, 55, 83].

For all parameters, we use standard CMOS scaling models to extrapolate to other technology nodes for evaluation.[10]

[10] 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 [54].

7.2 Validating the cost model

To validate our cost model (§5), we synthesize the full $\mathscr{P}$ logic for one layer of an arithmetic circuit of alternating addition and multiplication gates for several values of G. We also perform cycle-accurate RTL simulations for the same $\mathscr{P}$ logic, and record the pipeline delay (§5, Fig. 5) and the number of invocations of field addition and multiplication.[11]

Fig. 8 compares the data obtained from synthesis/simulation to that predicted by our model. Our 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.

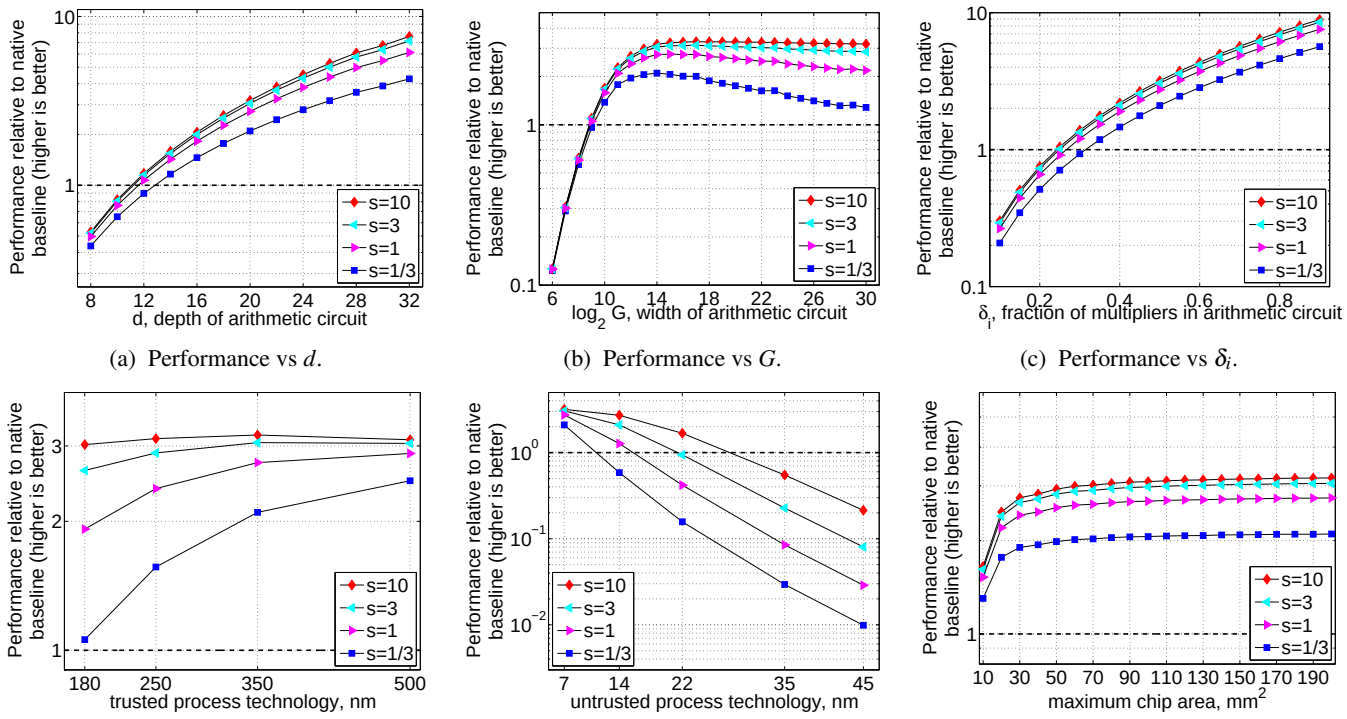
Although we have not presented a similar validation of $\mathscr{V}$'s costs, our cost model has similar fidelity for $\mathscr{V}$.

7.3 Baseline: native trusted implementations

For a quasi-straightline AC parameterized with depth d, width G and a δ_i fraction of multiplication gates, we estimate the costs of a "reasonable" implementation in the trusted technology node. To do so, we devise an optimization procedure to minimize $E \cdot A_s/T$ cost (§2.1) for such a circuit under the constraint that total area is limited to some maximum value.

Optimization proceeds in two steps. In the first step, the procedure apportions area to multiplication and addition prim-

[11] 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 (implying negligible energy cost).



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

(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 $E \cdot A_s/T$ metric (§2.1), varying $\mathscr{C}$ parameters, technology nodes, and maximum chip area. In each case, we vary one parameter and fix the rest. Fixed parameters are as follows: 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 $\mathscr{C}$, $\delta_i = 0.5$; maximum chip area, $A_{\max} = 200$ mm^2. In all cases, designs follow the optimization procedures described in Sections 7.1–7.3.

itives, based on δ_i and on the relative time and area cost of multiplication and addition primitives. 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, with hand-optimization, to exploit the structure of particular ACs in order to improve upon this optimization procedure; our goal is a procedure that gives good results for a generic quasi-straightline AC. We note that our strategy is roughly similar to the one used by automated hardware design toolkits such as Spiral [69].

7.4 Zebra versus baseline

This section evaluates the performance of Zebra versus the baseline high-assurance implementation described immediately above, on the metric $E \cdot A_s/T$, as a function of $\mathscr{C}$ parameters (G, d, δ), and as a function of technology nodes and maximum allowed chip size. We vary each of these parameters, one at a time, fixing others. Fixed parameters are as follows: $d = 20$, $G = 2^{14}$, $\delta_i = 0.5$, trusted technology node = 350 nm, untrusted technology node = 7 nm, and $A_{\max} = 200$ mm^2.

We do not report power dissipation, though it is a concern for physical realizability (§2.1). However, when Zebra is competitive with the baseline (on EAsT), its power dissipation is lower, since Zebra reduces energy consumption by more than it increases throughput (Figs. 12a, 12c; §8).

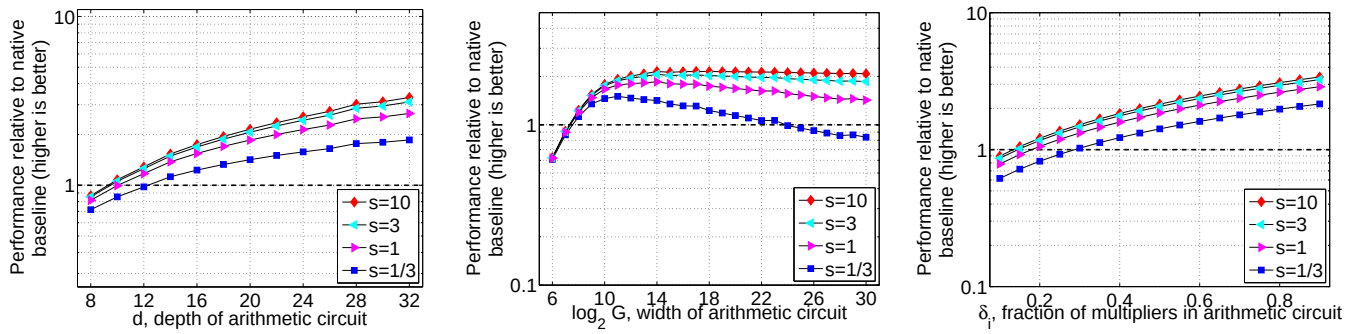
At the outset of an experiment, we optimize $n_{\mathscr{V},\text{io}}$, $n_{\mathscr{V},\text{sc}}$, and $n_{\mathscr{P},\text{sc}}$ (§5, Fig. 5). To do so, we first set $n_{\mathscr{V},\text{io}}$ and $n_{\mathscr{V},\text{sc}}$ to balance delay among $\mathscr{V}$'s layers (§3.5), and to minimize these delays subject to area limitations. We then choose $n_{\mathscr{P},\text{sc}}$ so that $\mathscr{P}$'s pipeline delay is no greater than $\mathscr{V}$'s.

Figure 9 summarizes the results. In each plot, the break-even point is designated by the dashed line at 1. We vary $s \in \{1/3, 1, 3, 10\}$; this is the downweighting factor for A_s (i.e., $A_s = A_{\text{trusted}} + A_{\text{untrusted}}/s$; §2.1).

We observe the following trends:

- As $\mathscr{C}$ grows in size (Figs. 9a, 9b) or complexity (Fig. 9c),[12] Zebra's performance improves compared to the baseline. This is because, for small circuits or those of low complexity, $\mathscr{V}$'s savings are not significant compared to the overhead of executing the protocol (§2.2).
- As the performance gap between the trusted and untrusted technology nodes grows (Figs. 9d, 9e), Zebra becomes increasingly competitive with the baseline. This is because Zebra relies on the technology gap to defray the high cost, for $\mathscr{P}$, of participating in the protocol.
- Figure 9b illustrates a tension between $\mathscr{V}$'s savings versus the baseline on complex circuits and $\mathscr{P}$'s work over such circuits: as $\log G$ grows, $\mathscr{P}$'s costs increase dramatically (§5, Fig. 5), resulting in increased $\mathscr{P}$ area. This is demonstrated by 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).

[12] Recall from Fig. 6 that multiplication is much more expensive than addition.



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

FIGURE 10—Zebra-lazy performance relative to lazy re-execution baseline (§7.5) on $E \cdot A_s$ metric (§2.1, §7.5), varying $\mathscr{C}$ parameters. In each case, we vary one parameter and fix the rest. Fixed parameters are as follows: 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 $\mathscr{C}$, $\delta_i = 0.5$; maximum chip area, $A_{\max} = 200$ mm^2.

7.5 Zebra versus other approaches

How do Zebra's costs compare to other approaches to verifiable execution? We consider two alternatives to the baseline of Section 7.3: (1) execution on untrusted hardware, followed by *lazy* re-execution on trusted hardware at a later point[13]; and (2) evaluating $\mathscr{C}$ on a trusted general purpose CPU (instead of an ASIC).

Execute now, check later. To evaluate the first alternative, we compare to an alternative instantiation of Zebra, which we call *Zebra-lazy*. In Zebra-lazy, outputs are computed as quickly as possible, on the same chip as $\mathscr{P}$; later—on the same time scale as alternative baseline (1)—$\mathscr{P}$ convinces $\mathscr{V}$ that this answer was correct, using the same machinery as Zebra.

Note that both Zebra-lazy and lazy re-execution yield results of computations at the same rate, namely, the speed of the untrusted chip. Thus, we compare these solutions using a simplified metric, $E \cdot A_s$, where we further stipulate that both solutions occupy the same area in the trusted technology node (note that we still charge Zebra-lazy for untrusted area).

For this comparison, we fix parameters as in Section 7.4, and we measure $E \cdot A_s$ versus $\mathscr{C}$ parameters d, G, and δ_i. Figure 10 summarizes the results. Zebra-lazy is still competitive with the lazy re-execution baseline, but the two systems are closer in performance. This is because, especially for "hard" computations (i.e., deep arithmetic circuits comprising mostly multiplications), Zebra gains a throughput advantage (and conversely on arithmetic circuits comprising mostly additions, it takes a speed penalty). Meanwhile, throughput is not part of the metric for the lazy setting.

Execute on trusted CPU. As an alternative to producing an ASIC in a trusted technology node, a principal might instead manufacture a general-purpose CPU for high-assurance execution, or even purchase older CPUs that the principal deems trustworthy (§2.1). A detailed analysis of these options requires addressing architecture and integration issues (like those in Section 4); but in general, because CPUs are far less efficient than purpose-built ICs, this approach will not yield a system that is comparable with Zebra on the $E \cdot A_s/T$ metric. Of course, other concerns (e.g., simplicity or ease of development) may militate in favor of the trustworthy CPU option; further treatment of these trade-offs is future work.

[13] This scenario obtains, among others, when the principal is able to take compensating measures after discovering that the untrusted hardware executed incorrectly. For example, a bank that discovers an incorrectly-processed transaction can coordinate with other banks to revert the transaction.

7.6 What is the price of verifiability?

A remaining high-level question is: what is the price of verifiability? It's clear from our evaluation that it is high: after all, Zebra only beats implementation in a trusted technology node when assisted by a far more advanced untrusted node, and only for sufficiently complex computations. Still, we regard this as progress: Zebra enables better performance than a native baseline, while preserving the latter's trustworthiness.

8 Applying Zebra to the NTT

This section evaluates Zebra on a specific computation of interest: the number theoretic transform over $\mathbb{F}_p$.

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

The Number Theoretic Transform is an algorithm closely related to the FFT that takes as input, the coefficients, $a_0, \ldots, a_{n-1}$, of a degree $n-1$ polynomial $p(x)$, and returns the tuple $p(\omega^0), \ldots, p(\omega^{n-1})$, where ω is a primitive n^{th} root of unity in the field $\mathbb{F}_p$. This transform is used prolifically in computer algebra packages and in cryptographic algorithms such as the SWIFFTX hash function [16]. Given its use in security-critical cryptographic applications, it is a suitable candidate for verifiable execution in hardware.

We implemented an iterative version of the NTT in Zebra using the standard sequence of butterfly operations [41]. This is convenient because this algorithm, which is the fastest known, is essentially given as an arithmetic circuit. However, this arithmetic circuit induces overhead when naively implemented in Zebra; we next refine OptimizedCMT to reduce that overhead.

8.2 Expanding Zebra's arithmetic circuit toolbox

The arithmetic circuit for NTT directly encodes the required butterfly operations. Each butterfly operation 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 $i, j \in \{0, \ldots, n-1\}$.

Recall from Section 2.2 that OptimizedCMT supports arithmetic circuit gates corresponding to $+$ and $\times$. While this is sufficient for computing the butterfly operation, it is inefficient: computing the difference $x_1 - x_2$ requires first computing $-1 \times x_2$. This increases the number of gates in each butterfly circuit; more importantly, it increases the depth of the AC, which increases both E (because it requires additional sumcheck invocations) and A_s/T (because performing these additional sumcheck invocations either requires more computational units, or increases the work for existing ones).[14]

To address this limitation, we follow an observation by CMT [42, §3.2] and introduce a small refinement to OptimizedCMT: direct support for subtraction gates. For $\mathscr{P}$, this change requires adding a new type of gate prover (§3.1). For $\mathscr{V}$ there are two small changes. First, we define a new wiring predicate (§2.2), sub_i, and its multilinear extension $\tilde{\text{sub}}_i \colon \mathbb{F}^{3b} \to \mathbb{F}$. During $\mathscr{V}$'s final computation at the end of each sumcheck round (line 100, Fig. 2), $\mathscr{V}$ adds an additional term $\tilde{\text{sub}}_i(q_{i-1}, w_0, w_1)(h_0 - h_1)$ accounting for the new gates. Second, $\mathscr{V}$ must retrieve from its private memory one additional precomputed value per round (§4).

This modification to Zebra mitigates overhead associated with the butterfly operations, at the cost of a small, constant amount of additional computation and storage per layer of $\mathscr{C}$.

8.3 Evaluation

Baseline and assumptions. The baseline is a direct implementation of the iterative arithmetic circuit for the NTT, optimized for $E \cdot A_s/T$ as described in Section 7.3. Our direct implementation is similar to hardware generated by the Spiral toolkit [69], a package for generating optimized hardware implementations of linear transforms.

In optimized NTT implementations, computing powers of ω is often inexpensive because ω can sometimes be chosen as a power of 2 (and thus multiplication by ω is just a bit shift). To model this optimization, we conservatively do not charge for computing powers of ω in our baseline.

On the other hand, for a length-n NTT, Zebra must either take $n/2$ powers of ω as inputs, or compute these values within $\mathscr{C}$. Because $\mathscr{V}$'s costs scale linearly with the number of inputs but only logarithmically with the size of each layer of $\mathscr{C}$ (§5), computing powers of ω in $\mathscr{C}$ seems to be the natural approach—but a naive implementation dramatically increases the depth of $\mathscr{C}$, erasing any savings. We observe that only 2^{i-1} distinct powers of ω are needed before the i^{th} butterfly operation. Thus, by providing as inputs only the $\log n$ powers $\{\omega, \omega^2, \omega^4, \omega^8, \ldots\}$, and by constructing each layer of $\mathscr{C}$ to compute powers of ω just as they are needed, we reduce the number of inputs to $\mathscr{C}$ from $3n/2$ to $n + \log n$ with no increase in depth, improving $E \cdot A_s/T$ by 25% or more when $\log n \geq 10$.

[14] In Section 7.4, increasing d improved the performance of Zebra relative to the baseline; but in that case, depth increased for *both* Zebra and the baseline. Here the restriction to $+$ and $\times$ comes from OptimizedCMT, and does not apply to the baseline, so baseline d would not increase.

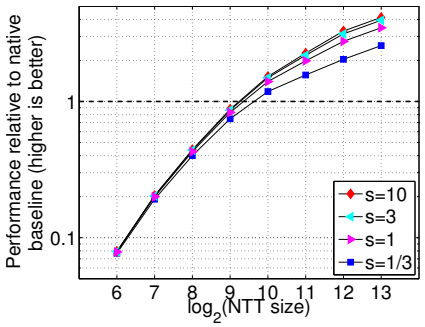


FIGURE 11—Zebra performance relative to baseline (§7.3, §8.3) on $E \cdot A_s/T$ metric (§2.1), for number theoretic transform with a given number of points, over $\mathbb{F}_p, p = 2^{63} + 2^{19} + 1$. Untrusted technology node = 350 nm; trusted technology node = 7 nm; maximum chip area, $A_{\max} = 200$ mm^2.

Method and results. We now evaluate Zebra versus the baseline described immediately above on the metric $E \cdot A_s/T$, for length-n NTTs over $\mathbb{F}_p, p = 2^{63} + 2^{19} + 1$.[15] As in Section 7.4, we fix trusted technology node = 350 nm, untrusted technology node = 7nm, and $A_{\max} = 200$ mm^2. We consider the cost weighting parameter $s \in \{1/3, 1, 3, 10\}$, and vary $\log n \in \{6, \ldots, 13\}$.

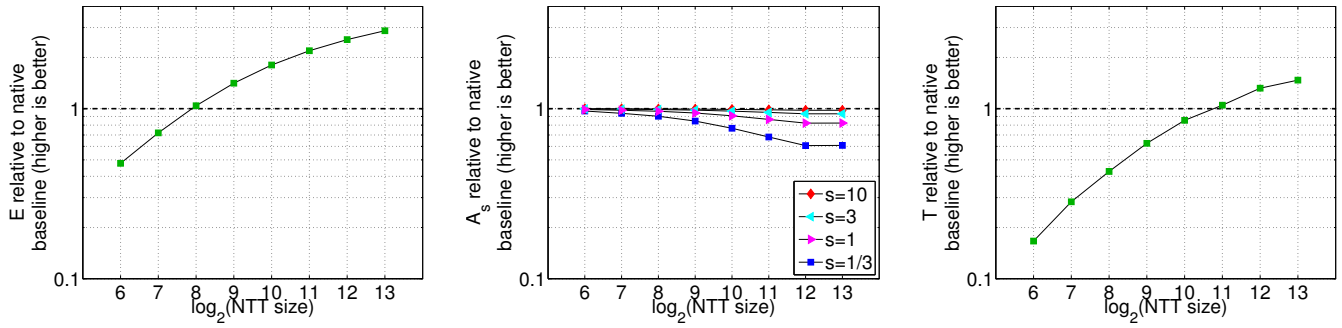
For the baseline, we compute $E \cdot A_s/T$ when area and pipeline stages are optimally partitioned subject to $\mathscr{C}$ and area limitations. For Zebra, we generate Verilog for $\mathscr{P}$ and $\mathscr{V}$ using our compiler toolchain (§6) and run cycle-accurate RTL simulations for sizes up to $\log n = 6$ using Cadence Incisive [2].[16] From these simulations we obtain (1) exact cycle counts (throughput), (2) exact field operation counts (energy), and (3) exact field arithmetic module counts (area); we use these to confirm that our cost model is still correctly calibrated for the new field (§5, §7.2). We use the cost model to compute $E \cdot A_s/T$ for larger n.

Figure 11 shows Zebra's performance compared to the baseline. Consistent with the results of Section 7.4, Zebra is not competitive for small computations, but it equals or exceeds the baseline when $\mathscr{C}$ is large enough.

To better understand *why* Zebra wins, we also compare Zebra and the baseline on each of E, A_s, and T individually. Figure 12 illustrates the results. At large NTT sizes, Zebra improves over the baseline on both E and T (Figs. 12a, 12c); in other words, for large computations, $\mathscr{V}$ executes fewer operations than the native baseline (this is consistent with the notion that $\mathscr{V}$ saves work by outsourcing; §2.2); and fewer operations executed translates to greater throughput. However, this advantage is diminished by the fact that the native computation is only about 30% multiplications (recall that this is a disadvantage for Zebra; §7.4); this is reflected in $\mathscr{V}$'s requiring larger NTT sizes to win on T (Fig. 12c). Finally, because Zebra and the baseline are constrained to the same trusted area (and $\mathscr{P}$'s area is nonzero), Zebra never wins on A_s (Fig. 12b); for larger NTTs (i.e., as

[15] We use a different field from §7.1 because the NTT requires a field with a subgroup of size 2^n; this requirement is fulfilled by this p for $n \leq 19$.

[16] This limit on simulation size reflects a practical issue: larger NTT sizes entail hours or days of simulation and massive amounts of RAM, even when using a highly optimized commercial Verilog simulator.



(a) Relative E vs. NTT size. (b) Relative A_s vs. NTT size. (c) Relative T vs. NTT size.

FIGURE 12—Individual performance comparisons of E, A_s, and T, for the number theoretic transform experiments (§8.3, Fig. 11).

$\mathscr{P}$'s area grows), A_s worsens compared to the baseline.

We conclude that Zebra applies to computations of interest, and gives materially better performance for E, T, and $E \cdot A_s/T$ than an optimized baseline, but uses more (untrusted) area.

9 Discussion

Zebra is not perfect. As our evaluation makes clear, its overhead is high (relative to untrusted execution), and it does not beat the baseline in all cases. On the other hand, the performance requirement, of including the prover in the cost when considering the break-even point, is extremely stringent (as noted earlier, prior works on verifiable computation work in a model in which only the verifier has to beat native execution). Furthermore, even if Zebra is *more* expensive than the baseline of executing directly on the trusted platform, it may still be warranted. Zebra's verifier can be fabbed once, or a small number of times (with different design parameters; §3, §5), and then the computation ($\mathscr{C}$) that it applies to can be configured *afterward* ($\mathscr{C}$ shows up in $\mathscr{V}$'s workload through the precomputed advice strings). Comparing the reuse of $\mathscr{V}$'s design and production to the fab-it-each-time requirement under the baseline, Zebra might be more attractive, its operational restrictions (§4) notwithstanding.

Moreover, as noted in the introduction, we expected to have to sacrifice something. Indeed, getting anything at all to work in hardware was challenging. We began this project by studying alternative verifiable computation machinery, based on GGPR [47], specifically Zaatar [74] and Pinocchio [71]. However, despite several months of work, we were unable to produce a physically realizable design that achieved reasonable throughput. The core reasons for this are severalfold. First, GGPR-based frameworks require expensive cryptographic operations, which are challenging to program in hardware. Second, the data flows are not local: the execution trace of the computation effectively has to be run through an FFT, and all parts of the proving algorithm have to touch it. As a third and related point, we found it difficult to extract the necessary parallelism in these frameworks.

Our experience building Zebra points to another question, namely, can we identify other protocols that are truly parallelizable *in practice*? Indeed, prior probablistic proof work is efficiently parallelizable in theory [27, §3], but our experience suggests that this does not immediately lead to a highly parallel hardware implementation. We leave further exploration of this question to future work.

10 Related work

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

Hardware Trojans. Defenses against hardware Trojans comprise Trojan *detection* and *disabling* techniques that operate either *post-fabrication* or *in-field*, and *deterrence* techniques at chip design time.

Post-fab techniques work within existing IC testing frameworks: they exercise the chip with certain inputs and verify that the outputs are correct. Wolff et al. [94] and Chakraborty et al. [37] propose augmenting 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), nor do they defend against Trojans designed to stay dormant during post-fabrication testing [79, 89], for instance, those activated by internal timers.

Other work proposes to detect changes in the delay and power of chips measured post-fabrication versus estimates obtained from pre-fabrication simulations [22, 59, 62, 63, 93]. However, the approach relies on the assumption 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. [92] exploit this uncertainty in the design of hardware Trojans that evade detection by such defenses.

Agrawal et al. [15] propose to extract "known good" delay and power profiles by destructively testing chips (meaning that they are depackaged, delayered, and optically imaged by a high resolution microscope) to establish that they are Trojan-free. However, such testing of large, complex ICs with billions of nanometer sized transistors is expensive, error-prone, and "stretches analytical capabilities to the limits" [84]. Furthermore, this approach is built on a restrictive assumption: that adversarial modifications to chips follow expected abstractions of "transistors" and "gates." Becker et al. [23] exploit this weakness to design stealthy Trojans that change only the doping

concentration of a few transistors in the chip; such Trojans are undetectable to optical imaging.

Waksman and Sethumadhavan [89] propose a *run-time* defense to *disable* Trojan triggers by power cycling the chip in the field (disabling timers), and to give the chip encrypted inputs (disabling in-built cheat codes). However, a limiting assumption in this work is that the chip can compute on encrypted inputs, which could have high cost. Moreover, an adversary can still use a randomly chosen input or sequence of inputs, or a chip aging sensor [92] to trigger an attack.

Another line of research is hardware obfuscation. Here, the aim is to protect intellectual property [36, 57, 73]. In addition, these projects propose to *deter* Trojans via obfuscation, the underlying assumption being that if the foundry cannot infer the chip's function, it cannot interfere with that function.

In contrast to all of the above approaches, Zebra defends against arbitrary misbehavior (under the assumptions of its setup), with a formal and comprehensive soundness guarantee.

A separate line of research focuses on mechanisms to detect Trojans inserted by a malicious designer before the chip is sent for fabrication [78, 90]; these techniques are complementary to Zebra's focus on an untrusted foundry.

Verifiable computation. The last several years have seen a flurry of work in built systems based on probabilistic proof protocols [20, 21, 24, 26, 28, 29, 32, 39, 42–45, 47, 61, 71, 74–76, 80, 82, 87, 88]. For our present purposes, these works fall into two categories. The first category descends from the Muggles [49] interactive proof (IP), and includes CMT [42], as well as Allspice [87], both of which Zebra builds on. The second category includes *arguments* of various kinds: interactive arguments with preprocessing [74–76] and *non-interactive* arguments with preprocessing [20, 26, 28–30, 39, 43, 47, 61, 71] (the non-interactive category is sometimes known as SNARKs, for succinct non-interactive arguments of knowledge).

The IP-based schemes offer information-theoretic security and, when they are applicable, have more efficient verifiers, with lower (or no) preprocessing costs. However, argument-based protocols handle a broader set of computations. In fact, recent works have extended arguments to handle ANSI C programs, RAM computations, cloud applications, set computations, databases, etc. [20, 26, 29, 32, 39, 43, 61, 88]. In addition, arguments offer 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 they make cryptographic assumptions (standard ones in the case of interactive arguments, non-falsifiable [70] ones in the case of SNARKs).

We were certainly inspired by this activity: as noted in Section 9, our initial objective was to implement SNARKs in hardware. However, implementing these protocols in hardware has proved challenging; to our knowledge, there are no prior hardware implementations of probabilistic proof protocols.

An intriguing middle ground is the GPU implementation of CMT, by Thaler et al. [82]. This implementation exploited some of the parallelism in CMT (as noted in Figure 2 and Section 3.2), and achieves speedups, versus CPU implementations of the prover and verifier. 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. [25], has previously articulated the goal of considering *both* the prover's and verifier's costs, when analyzing performance relative to the naive baseline. Their context is different: their paper 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 preprocessing and interaction. Nevertheless, this is inspiring work, because their result implies that, when considering another probabilistic proof framework, the combined overhead drops below the naive baseline asymptotically. However, the point at which this occurs in their setup—the "concrete-efficiency threshold" [25]—is problem sizes on the order of 2^{43}, which is larger than any of the aforementioned works can handle.

Acknowledgments

We thank Justin Thaler for a helpful optimization to GKR/CMT [81] and Greg Shannon for an inspiring conversation. Comments by Andrew J. Blumberg and Justin Thaler improved this draft. The authors of this work were supported by NSF grants CNS-1423249, CNS-1514422, CNS-0845811, TC-1111781, CNS-1527072, SRC-2015-TS-2635, AFOSR grant FA9550-15-1-0302, ONR grant N000141410469, the Microsoft Faculty Fellowship, and a Google Faculty Research Award.

References

[1] https://github.com/pepper-project.

[2] Cadence Incisive Enterprise Simulator. http://www.cadence.com/.

[3] Dell warns of hardware trojan. http://http://homelandsecuritynewswire.com/dell-warns-hardware-trojan.

[4] Icarus Verilog. http://iverilog.icarus.com/.

[5] IEEE standard 1800-2012: SystemVerilog. https://standards.ieee.org/findstds/standard/1800-2012.html.

[6] NanGate FreePDK45 open cell library. http://www.nangate.com/?page_id=2325.

[7] NCSU CDK. http://www.cs.utah.edu/~elb/cadbook/Chapters/AppendixC/technology_files.html.

[8] Protecting against gray market and counterfeit goods. http://blogs.cisco.com/news/protecting_against_gray_market_and_counterfeit_goods.

[9] Semiconductors: markets and opportunities. http://www.ibef.org/download/Semiconductors_220708.pdf.

[10] Tezzaron Semiconductor. "Can you trust your chips?". http://www.tezzaron.com/can-you-trust-your-chips.

[11] Trusted foundry program. http://www.dmea.osd.mil/trustedic.html.

[12] Christensen, Clayton and King, Steven and Verlinden, Matt, and Yang, Woodward. The new economics of semiconductor manufacturing. http://spectrum.ieee.org/semiconductors/design/the-new-economics-of-semiconductor-manufacturing.

[13] S. Adee. The hunt for the kill switch. http://spectrum.ieee.org/semiconductors/design/the-hunt-for-the-kill-switch, May 2008.

[14] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. In *IEEE S&P*, 2007.

[15] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan detection using IC fingerprinting. In *IEEE S&P*, May 2007.

[16] Y. Arbitman, G. Dogon, V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen. SWIFFTX: A proposal for the SHA-3 standard. NIST submission, 2008.

[17] B. M. Baas. A low-power, high-performance, 1024-point FFT processor. *IEEE Journal of Solid-State Circuits*, 34(3), Mar. 1999.

[18] L. Babai. Trading group theory for randomness. In *STOC*, May 1985.

[19] L. Babai, L. Fortnow, L. A. Levin, and M. Szegedy. Checking computations in polylogarithmic time. In *STOC*, May 1991.

[20] 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.

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

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

[23] G. T. Becker, F. Regazzoni, C. Paar, and W. P. Burleson. Stealthy dopant-level hardware trojans. In *Intl. Conf. on Cryptographic Hardware and Embedded Systems*, Aug. 2013.

[24] 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.

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

[26] 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.

[27] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza. Succinct non-interactive arguments for a von Neumann architecture. *IACR Cryptology ePrint Archive*, Dec. 2013. http://eprint.iacr.org/2013/879.

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

[29] 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. `http://eprint.iacr.org/2013/879/20140901:001903`.

[30] 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.

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

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

[33] 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.

[34] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli. Theory of latency-insensitive design. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 20(9), Sept. 2001.

[35] G. Carlson. Trusted foundry: the path to advanced SiGe technology. In *Compound Semicon. IC Symp.*, 2005.

[36] R. S. Chakraborty and S. Bhunia. HARPOON: an obfuscation-based SoC design methodology for hardware protection. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 28, Oct. 2009.

[37] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, and S. Bhunia. MERO: A statistical approach for hardware Trojan detection. In *Intl. Conf. on Cryptographic Hardware and Embedded Systems*, Sept. 2009.

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

[39] A. Chiesa, E. Tromer, and M. Virza. Cluster computing in zero knowledge. In *EUROCRYPT*, pages 371–403, Apr. 2015.

[40] S. Choi, R. Scrofano, V. K. Prasanna, and J.-W. Jang. Energy-efficient signal processing using FPGAs. In *FPGA*, Feb. 2003.

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

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

[43] 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.

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

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

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

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

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

[49] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum. Delegating computation: Interactive proofs for muggles. In *STOC*, May 2008.

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

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

[52] B. Grow, C.-C. Tschang, C. Edwards, and D. Burnsed. Dangerous fakes. *Business Week*, 2, 2008.

[53] L. Henzen, P. Gendotti, P. Guillet, E. Pargaetzi, M. Zoller, and F. Gürkaynak. Developing a hardware evaluation method for SHA-3 candidates. In *Intl. Conf. on Cryptographic Hardware and Embedded Systems*. Aug. 2010.

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

[55] 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 Journal of Solid-State Circuits*, 41(4), Apr. 2006.

[56] 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.

[57] 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.

[58] Y. Ishai, E. Kushilevitz, and R. Ostrovsky. Efficient arguments without short PCPs. In *Conference on Computational Complexity (CCC)*, 2007.

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

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

[61] 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.

[62] F. Koushanfar and A. Mirhoseini. A unified framework for multimodal submodular integrated circuits trojan detection. *IEEE Trans. on Information Forensics and Security*, 6(1), 2011.

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

[64] 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.

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

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

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

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

[69] P. Milder, F. Franchetti, J. Hoe, and M. Püschel. Computer generation of hardware for linear digital signal processing transforms. *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, 17(2), Feb. 2012.

[70] M. Naor. On cryptographic assumptions and challenges. In *CRYPTO*, pages 96–109, 2003.

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

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

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

[74] 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.

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

[76] 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.

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

[78] C. Sturton, M. Hicks, D. Wagner, and S. T. King. Defeating UCI: Building stealthy and malicious hardware. In *IEEE S&P*, pages 64–77, 2011.

[79] M. Tehranipoor and F. Koushanfar. A survey of hardware Trojan taxonomy and detection. *IEEE Design and Test of Computers*, 27(1), Jan. 2010.

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

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

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

[83] 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.

[84] R. Torrance and D. James. The state-of-the-art in IC reverse engineering. In *Intl. Conf. on Cryptographic Hardware and Embedded Systems*, Sept. 2009.

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

[86] T. Vogelsang. Understanding the energy consumption of dynamic random access memories. In *IEEE/ACM Symp. Microarchitecture*, Dec. 2010.

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

[88] 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.

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

[90] A. Waksman, M. Suozzo, and S. Sethumadhavan. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *ACM CCS*, pages 697–708, 2013.

[91] M. Walfish and A. J. Blumberg. Verifying computations without reexecuting them: from theoretical possibility to near practicality. *Communications of the ACM*, Feb. 2015.

[92] 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.

[93] S. Wei, S. Meguerdichian, and M. Potkonjak. Malicious circuitry detection using thermal conditioning. *IEEE Trans. on Information Forensics and Security*, 6(3), 2011.

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

[95] Q. Xie, Y. Wang, and M. Pedram. Variability-aware design of energy-delay optimal linear pipelines operating in the near-threshold regime and above. In *ACM Great Lakes Symp. on VLSI*, May 2013.

[96] 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.