ZSafe: Proving the Safety of Proprietary Hardware Designs in Zero Knowledge

ANONYMOUS AUTHOR(S)

Protecting the confidentiality of hardware designs is essential, especially during third-party verification, where proprietary designs must be examined by potentially untrusted verifiers. Existing approaches, such as obfuscation, watermarking, design encryption, and prior privacy-preserving verification techniques, are either not applicable, rely on trusted third parties, or lack scalable formal guarantees.

We present ZSafe, the first zero-knowledge (ZK) framework to provide formal safety guarantees for proprietary hardware designs. ZSafe enables a designer to prove the safety of confidential hardware to a verifier without revealing the design. Our approach introduces (i) an encoding scheme for hardware design, and a ZK-friendly constraint system that binds hardware design and its formula representation, (ii) an efficient ZKP protocol for validating logical implication, avoiding the prohibitive overhead of formula translation by using probabilistic checking, and (iii) a shared-structure unsatisfiability proof protocol to handle interrelated verification checks efficiently. We implement ZSafe and evaluate it on 69 adapted HWMCC'24 hardware verification benchmarks with up to 300,000 gates. The results demonstrate practical scalability: 90% of instances are proven within an hour.

CCS Concepts: • **Security and privacy → Logic and verification**; **Privacy-preserving protocols**; **Security in hardware**.

Additional Key Words and Phrases: Zero-knowledge proof, formal method, privacy-preserving hardware safety verification

ACM Reference Format:
Anonymous Author(s). 2026. ZSafe: Proving the Safety of Proprietary Hardware Designs in Zero Knowledge. In *Proceedings of (Conference OOPSLA '26)*. ACM, New York, NY, USA, 28 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The proprietary architectures, optimization strategies, and domain-specific logic embedded in modern hardware designs result from substantial and resource-intensive investment. Industry estimates suggest that in-house processor design often costs $100–200 million over 3–4 years [6, 55]. Such costs make intellectual property (IP) theft both highly tempting and a significant threat to the hardware design industry. The risk is amplified as companies increasingly rely on a fragmented network of external partners to acquire and integrate IP designs, making valuable IPs vulnerable to theft at every stage of development. In fact, the Semiconductor Industry Association (SIA) estimates that IP theft costs U.S.-based semiconductor companies more than $7.5 billion annually [52].

Verification has become one of the most vulnerable points for IP theft. Hardware verification is a critical step to ensure the design's safety and correctness, and has been extensively studied [15, 19, 28, 37, 41, 42, 59, 60]. However, these verification techniques require exposing the core proprietary logic to external partners. The dependency on third-party trust inherently carries a substantial risk

Conference OOPSLA '26, CA, USA

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

of IP misappropriation. Existing approaches via contractual safeguards and legal efforts alone have been shown to be insufficient for sophisticated cases [47–50].

Cryptographic approaches offer strong privacy guarantees [29, 42], but are limited to correctness checks on individual execution traces, thereby weakening the resulting guarantees. Meanwhile, the overhead of cryptographic primitives makes them prohibitive for practical hardware verification at scale. For example, Pythia [42] requires about 500 s to verify a small design with only a few thousand gates while preserving design confidentiality, far below the hundreds of thousands of gates found in real-world designs.

This paper introduces ZSafe, the first framework for protecting the confidentiality of hardware designs during verification. ZSafe considers two distinct participants: the **designer**, who holds the confidential design, and the **verifier**, who checks whether the designer's design satisfies a given safety property. ZSafe provides rigorous guarantees on confidentiality and soundness. The design remains private throughout verification and is never revealed to the verifier, achieved using cryptographic tools such as commitment schemes and zero-knowledge proofs (ZK). Meanwhile, if verification by ZSafe succeeds, the verifier is guaranteed that the safety property holds over every possible execution trace of the hardware, ensured by invariant-based inductive verification.

Inspired by classical hardware verification, ZSafe leverages inductive invariants to prove safety over a design's entire state space. An inductive invariant is a property over the state space that holds on all initial states, is preserved by every transition, and implies the target safety property. Given such an invariant, safety verification *reduces* to checking the *unsatisfiability* of constraint systems consisting of three formulas corresponding to initiation, transition, and safety implication.

In ZSafe, the designer first commits to the design, invariant, and formula using a cryptographic commitment scheme, which binds these values without revealing them to the verifier. ZSafe proposes an encoding scheme and arithmetization for hardware design to enable their efficient commitment. To implement invariant-based safety verification over commitments, ZSafe introduces efficient protocols for verifying the validity of *reductions* from design safety to formula unsatisfiability, without revealing the design or the formulas. Meanwhile, ZSafe generalizes prior work on privacy-preserving unsatisfiability checking for CNF formulas [39] to implication-form formulas used in safety-verification tasks, without incurring prohibitive overhead.

ZSafe introduces a polynomial-based encoding that serves as a unified algebraic bridge between hardware designs and their safety properties. The design encodings are constructed to capture two complementary dimensions of hardware behavior: the functional semantics of individual logic gates, and the temporal relationships among circuit states across execution steps. Furthermore, this encoding supports efficient polynomial commitments and verification in ZK, making ZSafe practical.

Building on this encoding framework, ZSafe introduces an efficient protocol for verifying the refinement relation between the hardware design and its formula representation in the privacy-preserving setting. To support this, ZSafe encodes verification formulas over the same polynomial domain as the hardware design. When a safety formula and the hardware design are aligned on their shared encoding symbols, determining whether the formula faithfully reflects the design's behavior reduces to checking a system of polynomial constraints, which is checked later in ZK so that neither the design nor the property is revealed.

Unsatisfiability checking for the committed formulas for safety verification requires translating them from implication form to CNF. The translation must operate over commitments to preserve confidentiality. ZSafe implements this via a protocol that probabilistically verifies the Tseytin translation in ZK. By having the verifier send random challenges to the designer, ZSafe avoids the exponential blowup in the formula size and the prohibitive verification cost of enforcing the variable-freshness requirement of Tseytin translation. To make hardware formal verification

in ZKPs practical, we augment ZSafe with two optimizations: Balanced Tseytin translation and strengthened invariants.

We implement a prototype of ZSafe in C++ and evaluate it on the publicly well-known Hardware Model Checking Competition (HWMCC'24) benchmark suite [3]. The benchmark set contains a diverse collection of verification tasks spanning communication protocols (e.g., bounded retransmission protocol), reactive control systems (e.g., elevator controllers and arbitration logic), signal-processing datapaths (e.g., FIR filters), graphics pipelines (e.g., VGA image FIFO controllers), and processors (e.g., picoRV and ZipCPU).

On an AWS r5b.16xlarge instance, ZSafe proves more than half of the 69 benchmark designs within 200 seconds and 90% within one hour. Furthermore, our optimization using strengthened invariants directly reduces one-third of the burden of necessary safety proofs. Balanced Tseytin translation accelerates verification for challenging benchmarks such as the bounded retransmission protocol, whose invariant contains 424 lemmas, reducing the verification time from 5.6 hours to 3 hours, a 45% reduction. We give a detailed evaluation in Section 4.3, which includes the end-to-end verification time for each instance.

In this work, we make the following contributions.

- We design a ZKP framework for proving safety properties of Boolean transition systems without revealing the system or inductive invariant.
- We introduce a polynomial-based encoding scheme that captures both the functional semantics of individual logic gates and the temporal evolution of circuit states across time steps. These encodings are compatible with efficient polynomial commitment schemes in cryptography.
- To avoid the prohibitive overhead introduced by verifying inductive proof in ZK, we propose a new ZK protocol that probabilistically verifies logical implications. Combined with clause consistency checks across inductive subformulas, this approach enables efficient and sound verification of inductive safety proofs for hardware design in ZK.
- We implement ZSafe and evaluate it on adapted HWMCC'24 hardware verification benchmarks to demonstrate scalability. ZSafe verifies 63 of 69 instances within an hour, proving formal inductive safety guarantees in ZK for hardware designs with up to 300,000 gates. Existing privacy-preserving hardware approaches [29, 42, 43] only check individual circuit executions rather than formal safety properties. Pythia [42] was evaluated on designs containing a few thousand gates by proving their execution in ZK. ZSafe proves stronger properties and supports designs that are orders of magnitude larger than those evaluated in prior work.

1.1 Our Goals and Threat Model

In this section, we describe our goals and threat model. ZSafe is built in a privacy-preserving pipeline that enables a designer to prove the safety of a proprietary hardware design to an untrusted verifier, as depicted in Figure 1. The designer begins by publishing a cryptographic hash h of the design, which binds the design to a public value without revealing it. Once the design's safety has been verified in a privacy-preserving manner, any party can obtain a safety guarantee simply by checking that the design hashes to the published value. Since verifying a hash value in the privacy-preserving manner is a well-established technique [4, 13, 25, 26], we do not address it further in this work.

ZSafe focuses on the following design objectives as the core of the pipeline: (1) encodings of the design and verification formulas that enable efficient commitment schemes; (2) practical protocols for verifying the correctness of the reduction from design safety to formula unsatisfiability, without revealing the design or the formulas; and (3) practical protocols for checking the unsatisfiability of the constraint system derived from the verification tasks in a privacy-preserving manner.

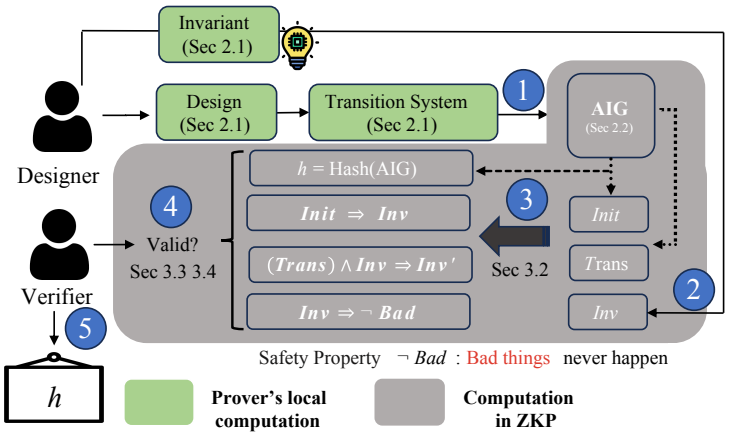


Fig. 1. ZSafe lies at the center of a privacy-preserving verification pipeline. (1) The designer models the hardware design as a transition system represented by an AIG, which is committed by ZSafe. (2) The designer computes a private inductive invariant based on a public safety property and also commits it. (3) Designer in ZSafe convinces the verifier, via ZK, that the transition relation $Trans$ is correctly constructed from the private AIG without revealing $Trans$ or the AIG. (4) The verifier in ZSafe validates, in ZK, three implication checks for safety verification without revealing the formula. (5) Finally, the verifier publishes a commitment (e.g., a hash) bound to the AIG, generated within the ZK, linking the proved safety properties to the proprietary design. In any downstream task that requires safety, the only thing needed is to check whether the hash of the design matches the published value h.

1.1.1 Threat Model. ZSafe considers a two-party setting between a designer and a verifier. The designer represents an IP seller seeking to license or sell a proprietary hardware design. The verifier represents an IP buyer who intends to integrate, manufacture, or deploy the design, but first must verify that the design satisfies specified safety properties. For example, a designer could deliver a backdoored circuit whose behavior changes after many cycles of execution. The verifier could use ZSafe to check that this does not happen without learning the hardware's proprietary design details.

We assume either party may behave maliciously, but do not consider collusion. A malicious verifier may attempt to infer confidential design information from the proving process, while a malicious designer may try to falsely convince the verifier of an unsafe design. ZSafe aims to enable the designer to convince the verifier of the safety of proprietary hardware designs, and any failure during the proving procedure can be attributed to either a design error or a violation of the safety requirements.

1.1.2 Information Leakage.
ZSafe ensures that the committed design and inductive invariants remain private to the designer. The verifier only learns the public safety property, whether the design satisfies the safety property, and upper bounds on the number of AND/LATCH gates in the private design and on the size of the inductive invariant. ZK execution time and proof size only depend on these public parameters. ZSafe reveals no additional information to the verifier.

1.1.3 Non-goal. We focus specifically on preserving the confidentiality of proprietary hardware IP during the verification process. IP theft that occurs prior to or following verification, such as unauthorized redistribution and hardware reproduction, falls outside the scope of this work. The related work is discussed in Section 6.2.

2 Preliminaries

2.1 Hardware Safety Verification

2.1.1 Propositional Boolean Formula. A propositional Boolean formula is a logical expression composed of Boolean variables and logical operators such as conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). It evaluates to either true or false under a given assignment of truth values to its variables. A formula is *unsatisfiable* if there is no assignment of truth values to its variables that makes the entire formula evaluate to true.

A Boolean formula is said to be in Conjunctive Normal Form (CNF) if it is expressed as a conjunction of clauses, where each clause is a disjunction of literals in $\mathbb{L}$ (Boolean variables or their negations). The width of a clause refers to the number of literals it contains. Moreover, a CNF formula $\phi = \bigwedge_{i=1}^{n} C_i$ can be equivalently represented as a set of clauses $C_\phi = \{C_1, C_2, \ldots, C_n\}$.

2.1.2 Transition system. A transition system models the behavior of a hardware design and is defined as $\mathcal{T} = (\mathtt{Var}, \mathit{Init}, T)$, where $\mathtt{Var} = \{v_1, v_2, \ldots, v_n\}$ is a finite set of state variables usually corresponding to the registers and latches in the circuit. A state $\mathtt{s} \in \mathtt{S}$ is a complete valuation of all state variables, representing a snapshot of the hardware. $\mathit{Init}(\mathtt{s})$ is a predicate defining the set of initial states. $T(\mathtt{s}, \mathtt{s}')$ is a transition relation encoding how the hardware evolves from a current state $\mathtt{s}$ to a next state $\mathtt{s}'$.
A finite execution trace of $\mathcal{T}$ is a sequence of states $\mathtt{s}_0, \mathtt{s}_1, \ldots, \mathtt{s}_k$ such that $\mathit{Init}(\mathtt{s}_0)$ holds and $T(\mathtt{s}_i, \mathtt{s}_{i+1})$ holds for all $0 \leq i < k$.

2.1.3 SAT-based induction proof of safety. A safety property, which states that "something bad never happens," can be reduced to a *reachability* check. That is, given a transition system $\mathcal{T} = (\mathtt{Var}, \mathit{Init}, T)$ and a bad state predicate *Bad*, the safety is violated if there exists a finite path $\mathtt{s}_0, \ldots, \mathtt{s}_k \in \mathtt{S}^k$ such that $\mathtt{s}_0 \models \mathit{Init}$, $(\mathtt{s}_i, \mathtt{s}_{i+1}) \models T$ for $0 \leq i < k$, and $\mathtt{s}_k \models \mathit{Bad}$. Verification of the unreachability of bad states can be performed using an inductive proof by identifying an *inductive invariant* II that over-approximates the set of all reachable states of the transition system, such that:

$$\mathit{Init}(\mathtt{s}) \implies \mathit{II}(\mathtt{s}) \qquad \text{(initialization)}$$
$$\mathit{II}(s) \wedge T(\mathtt{s}, \mathtt{s}') \implies \mathit{II}(\mathtt{s}') \qquad \text{(transition)}$$
$$\mathit{II}(s) \implies \neg \mathit{Bad}(\mathtt{s}) \qquad \text{(safety implication)}$$

The inductive proof consists of three steps: 1) the *initialization* shows that the inductive invariant II holds for all initial states; 2) the *transition* proves that II is preserved under the transition relation, meaning that if $\mathit{II}(\mathtt{s})$ holds and $\mathtt{s}'$ is a successor of $\mathtt{s}$ (i.e., $(\mathtt{s}, \mathtt{s}') \models T(\mathtt{s}, \mathtt{s}')$), then $\mathit{II}(\mathtt{s}')$ also holds; 3) the *safety implication* ensures that no state satisfying II is a bad state. Together, these checks establish that $\mathcal{T}$ has no transition path leading to any state satisfying *Bad*. That is to prove the validity of the following formula, or equivalently, the unsatisfiability of their negation:

$$\mathit{Init}(\mathtt{s}) \wedge \neg \mathit{II}(\mathtt{s}), \quad \mathit{II}(\mathtt{s}) \wedge T(\mathtt{s}, \mathtt{s}') \wedge \neg \mathit{II}(\mathtt{s}'), \quad \mathit{II}(\mathtt{s}) \wedge \mathit{Bad}(\mathtt{s})$$

2.2 Sequential And-Inverter Graph

A sequential And-Inverter Graph (AIG) is a specialized Boolean network [51] representing sequential logic circuits, defined as $C_{\text{AIG}} = (V, E, I, L, O, A, \mathit{inv}, \mathit{init})$, where:

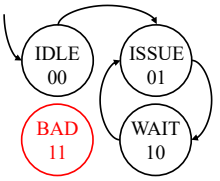
- V is the set of all vertices, partitioned into primary inputs $I \subseteq V$, latches $L \subseteq V$, primary outputs $O \subseteq V$, and 2-input AND gates $A \subseteq V$. $E \subseteq V \times V$ is the set of directed edges. The signal σ_v denotes the Boolean output of vertex $v \in V$.

```
module controller (
  input wire clk,
  input wire ack, // acknowledge
  output wire req // request
);
reg r1 = 1'b0;// state = {r1,r0}
reg r0 = 1'b0;
always @(posedge clk) begin
  r0 <= (~r1 & ~r0) | ( r1 & ack);
  r1 <= (~r1 & r0) | ( r1 & ~ack);
end
assign req =~r1 & r0;
endmodule
```

(a) A Verilog implementation of a handshake controller with two state registers r_0 and r_1.

Literal	Variable	Meaning
2	v_1	ack (input)
3		$\neg$ack
4	v_2	r_0 (latch, next = lit 16)
5		$\neg r_0$
6	v_3	r_1 (latch, next = lit 18)
7		$\neg r_1$
8	v_4	$n_1 = \neg r_1 \wedge \neg r_0$
10	v_5	$n_2 = r_1 \wedge$ ack
12	v_6	$n_3 = \neg r_1 \wedge r_0$
14	v_7	$n_4 = r_1 \wedge \neg$ack
16	v_8	$n_5 = \neg n_1 \wedge \neg n_2$
18	v_9	$n_6 = \neg n_3 \wedge \neg n_4$
20	v_{10}	req = $\neg r_1 \wedge r_0 = n_3$

(b) The design can be represented as an AIG circuit consisting solely of AND and LATCH gates.



(c) The controller transitions from Issue to Wait, and returns to Issue upon receiving ack. State *Bad* is unreachable by the execution of the handshake controller.

Fig. 2. An example hardware design, its AIG representation, and the corresponding transition system. The two-bit state (r_1, r_0) encodes four states: Idle (00), Issue (01), Wait (10), and Bad (11), where Bad corresponds to both registers being asserted simultaneously. The safety property requires that the Bad state is unreachable in any execution of the design.

- $inv : E \longrightarrow \{0, 1\}$ is an inversion map on edges. For each edge $(u, v) \in E$, the signal received by v from u is σ_u if $inv(u, v) = 0$ (no inversion), or $\neg\sigma_u$ if $inv(u, v) = 1$ (inverted). This allows negation to be embedded directly on edges rather than requiring explicit inverter nodes.
- $init : L \longrightarrow \{0, 1\}$ defines the initial output value of each latch.

For a 2-input AND gate $a \in A$ with predecessors $u_1, u_2 \in V$, its output is computed as $\sigma_a = \neg^{inv(u_1,a)}\sigma_{u_1} \wedge \neg^{inv(u_2,a)}\sigma_{u_2}$. For a latch $v \in L$ with predecessor $u \in V$, its output is initialized to $init(v)$ and updated each clock cycle as $\sigma_{v'} = \neg^{inv(u,v)}\sigma_u$, where $\sigma_{v'}$ denotes the next-state output of latch v.

2.3 AIG Construction

Given a transition system $\mathcal{T} = (\mathtt{Var}, Init, T)$ over Boolean state variables $\mathtt{Var} = \{v_1, \ldots, v_n\}$, we can construct a sequential AIG $C_{\text{AIG}} = (V, E, I, L, O, A, inv, init)$.

- For each state variable $v_i \in \mathtt{Var}$, introduce a latch $l_i \in L$, so that $L = \{l_1, \ldots, l_n\}$ and σ_{l_i} represents v_i.
- For each latch $l_i \in L$, set $init(l_i) = 1$ if $Init$ requires $v_i = 1$ in the initial state, and $init(l_i) = 0$ otherwise.
- For each latch $l_i \in L$, introduce a combinational subcircuit over A and I that computes the next-state function $v_i' = f_i(v_1, \ldots, v_n)$ as defined by T, using AND gates and edge inversions.
- The primary inputs I encode any non-determinism in T, i.e., free variables not determined by the current state.
- The primary output O encodes the safety property $\neg Bad(\mathtt{s})$ to be verified.

The resulting AIG satisfies $\mathtt{Var} = \{\sigma_l \mid l \in L\}$, and the transition relation $T(\mathtt{s}, \mathtt{s}')$ is faithfully captured by the combinational logic driving the next-state inputs of the latches. Figure 2 shows an AIG for the handshake controller design specified in Verilog and its transition system.

2.4 Zero-Knowledge Proof

Zero-knowledge proof (ZK) [22, 24] allows a prover to convince a verifier that it knows a witness w such that a public predicate $P(x, w) = 1$ holds for some public input x, without revealing any information about w beyond the validity of the statement. In addition, the verifier will determine, with very high confidence, whether a prover is dishonest and the statement is false. There have been

many lines of work in designing practically efficient ZK protocols [23, 27, 33, 34] under different settings and assumptions.

Commit and prove. A commitment scheme allows a party to commit to a value without revealing it, while binding them to that value for future verification. Commit-and-prove ZK systems allow one party (prover, or the designer in ZSafe) to first commit to secret values. Then the prover can prove, and the verifier can verify that those committed values satisfy a relation in ZK, without revealing the values themselves.

Polynomial equivalence. Given two private polynomials $P_L(X)$ and $P_R(X)$ over a finite field $\mathbb{F}$, existing ZKP protocols enable proving that $P_L(X)$ and $P_R(X)$ are equivalent. Private polynomials mean polynomials whose coefficients are private. We consider only private polynomials with a publicly bounded degree.

Subset checking. Given a private set C_{sub} and a collection of private sets $C_{\text{sup}1}, \ldots, C_{\text{sup}k}$, existing ZKP protocols enable the prover to prove that $C_{\text{sub}} \subseteq C_{\text{sup}1} \cup \cdots \cup C_{\text{sup}k}$. Private sets are bounded-size sets whose elements are private and drawn from a public domain.

Unsatisfiability checking. Our work leverages the existing ZKP protocol of the CNF formula's unsatisfiability. ZKUNSAT [39] proves the unsatisfiability of a propositional Boolean formula without revealing the formula itself. It is based on a ZKP-friendly arithmetization of the resolution proof verification. The unsatisfiability checking is reduced to checking the relationship over polynomials over a finite field. Such polynomial relation checking is then implemented in the ZKP backend, ensuring the privacy of the formula.

Literal encoding in ZKUNSAT relies on a mapping $\epsilon : \mathbb{L} \mapsto \mathbb{F}$. ZKUNSAT encodes each literal $\ell \in \mathbb{L}$ as a field element $\epsilon(\ell) \in \mathbb{F}$. To ensure correctness, the mapping is required to be injective and to satisfy, for a fixed public constant $\alpha \in \mathbb{F}$,

$$\epsilon(\ell_i) + \epsilon(\neg \ell_i) = \alpha.$$

A clause $C = \ell_0 \vee \cdots \vee \ell_{d-1}$, where each $\ell_i \in \mathbb{L}$, is encoded as a univariate polynomial whose roots correspond to the field encodings of the participating literals:

$$\gamma_\epsilon(C)(X) = \prod_{i=0}^{d} (X - \epsilon(\ell_i)) .$$

Let C_ϕ be the set of clauses in the CNF formula ϕ. Then ϕ can be encoded as a set of polynomial encodings of all clauses. That is, the encoding of ϕ is given by the set $\left\{\gamma_\epsilon(C_i)(X) \,\middle|\, C_i \in C_\phi\right\}$.
When a CNF formula ϕ is encoded as the above, ZKUNSAT enables us

- verify the unsatisfiability of a private CNF ϕ in ZKP;
- verify a private clause $C \in C_\phi$ for a private CNF ϕ without revealing ϕ.

Remark. The time cost of ZKUNSAT is linear in *the number of involved clauses* multiplied by the *maximum clause width*. In addition, introducing new variables into ZKUNSAT is free if $|\mathbb{F}| \geq |\mathbb{L}|$. That is, there exists an injective mapping from $\mathbb{L}$ to $\mathbb{F}$ or C.

3 ZSafe

ZKUNSAT provided an efficient ZK polynomial encoding of CNF unsatisfiability. This is insufficient for hardware verification because it does not bind formulas to a committed sequential AIG or enforce current/next-state and invariant consistency across inductive obligations. Throughout this section, we present ZSafe which addresses these limitations. ZSafe is our framework for privacy-preserving safety verification of proprietary hardware designs. We begin by outlining the overall system architecture and workflow of ZSafe. We then describe three key technical components in detail:

(1) The algebraic encoding of the hardware design, which enables ZK-friendly representation of hardware design semantics;
(2) A new ZK protocol for verifying logical implications required by inductive safety proofs, which avoids the overhead of formula format translation;
(3) The consistency checking of subformulae in ZK, ensuring that different checks in the safety proof refer to the same underlying components without revealing any confidential information.

3.1 ZSafe Overview

The input to ZSafe involves two parties: the designer and the verifier. The designer provides (1) private hardware design, formulas, and inductive invariants. The safety property to be verified is public and known to both parties. At the end of the protocol, the verifier outputs a single bit indicating whether the safety property holds for the design, without learning any information about the hardware design. The hardware design is represented as a sequential AIG, encoded as a list of tuples over the finite field $\mathbb{F}$, where each tuple corresponds to either an `AND` gate or a `LATCH` element in the circuit. Formulas, invariants, and safety properties are expressed in CNF and are further encoded as sets of polynomials as described in Section 2.4.

The designer first commits the hardware design, the CNF formula for initialization *Init*, transition T, and inductive invariants Π (described in Section 2.1). To enable commitment over hardware design and verify the correctness of *Init* and T for it, ZSafe introduces a new encoding scheme for AIG. This encoding scheme reduces the correctness of formula construction to a set of polynomial-equivalence constraints, which are then verified by ZK backends. Details are provided in Section 3.2.

To demonstrate that the design satisfies a safety property, ZSafe also proves the validity of formula encoding initialization, transition, and safety implication (equivalently, unsatisfiability of their negation, Section 2.1). However, since the formulas in this constraint system are in implication form rather than CNF, existing ZK unsatisfiability backends cannot be directly applied. Note that only *Init*, T, and Π are committed by the designer in CNF. We detail our approaches to verifying the unsatisfiability of the formula in implication form (Section 3.3) and enforcing clause consistency across formulas (Section 3.4).

ZSafe introduces two optimizations to reduce ZK verification cost. First, rather than applying the standard Tseytin translation, ZSafe leverages a *Balanced Tseytin Translation* to enforce a uniform bound of $\omega + 1$ on clause width, significantly reducing the computation cost for unsatisfiability checks. Second, ZSafe strengthens the inductive invariant by conjoining the negation of the bad-state predicate, $\widehat{\Pi}(s) = \Pi(s) \wedge \neg Bad(s)$, eliminating the safety implication check and avoiding the conversion overhead in ZK for the safety property.

3.2 Formulas Correctness Verification

The designer commits to a sequential AIG encoding the hardware design at the bit level, along with CNF formulas *Init* and T representing the initialization and transition relation of the induced transition system. A malicious designer can commit *Init* and T that deviate from the design's execution, thereby cheating about the design's safety. To ensure soundness, the designer in ZSafe first proves to the verifier that the committed formulas *Init* and T faithfully capture the design's behavior, using only the commitments from the designer.

In this section, we first describe the encoding and arithmetization of the AIG, where its structure is encoded as a list of elements over a finite field $\mathbb{F}$, along with their relations. When the designer commits the design, she commits such an arithmetization of this design. This encoding converts the AIG into algebraic data compatible with formula encoding in ZK protocols we use later. We then introduce polynomial constraints that bind the AIG to the formula, which are verified in ZK

as described in 2.4, without revealing all polynomials. By verifying that the committed formula and design encodings satisfy these polynomial constraints in ZK, ZSafe enforces the correctness of committed $(Init, T)$. Our constraints are carefully tailored and can be efficiently verified in ZK, ensuring the practical performance of ZSafe.

3.2.1 Timestamped Signal Encoding. For a signal $\sigma \in \Sigma$ in the AIG, we define two distinct encodings, one for the current state and one for its next state. In particular, we leverage the following injective maps ϵ and ϵ^T from signal sets Σ to a finite field $\mathbb{F} \setminus \{\gamma_\perp, \gamma_\top\}$, satisfying for any $\sigma \in \Sigma$:

$$\epsilon(\Sigma) \cap \epsilon^T(\Sigma) = \emptyset \tag{1a}$$

$$\epsilon(\sigma) + \epsilon(\neg\sigma) = \epsilon^T(\sigma) + \epsilon^T(\neg\sigma) = \gamma_\perp + \gamma_\top = \alpha \tag{1b}$$

$$\epsilon^T(\sigma) = \beta + \epsilon(\sigma) \tag{1c}$$

The intention behind this construction is that ϵ and ϵ^T represent encodings of the same signal in two consecutive states of the transition system. We now explain each requirement for our encoding:

- Equation 1a ensures that the encodings of signals before and after the transition do not overlap;
- Equation 1b satisfies the encoding requirements imposed by ZKUNSAT for both ϵ and ϵ^T;
- Equation 1c binds the encodings of the same signal across the transition.

3.2.2 AIG Encoding. Each gate in the AIG, whether an AND gate or a LATCH, is encoded as a tuple of three field elements derived from the signal encoding. The designer commits the AIG by committing to each element of every such tuple independently.

- Each two-input AND gate with input signals σ_a, σ_b and output signal σ_c is represented by the triple $\big(\epsilon(\sigma_a),\ \epsilon(\sigma_b),\ \epsilon(\sigma_c)\big)$.
- Each latch gate with input signal σ_i and output signal σ_o is represented by the triple $\big(\epsilon(\sigma_i),\ \epsilon(\sigma_o), Init(v)\big)$, where $Init(v)$ denotes the initial value of the output of latch v.

3.2.3 Correctness verification of $T(\mathsf{s}, \mathsf{s'})$. Each tuple of a gate g induces a set of polynomials S_g, described in the next section, that can be directly constructed from the tuple elements. To prove the correctness of a formula, the designer must show that every clause in the formula $T(\mathsf{s}, \mathsf{s'})$ corresponds to some gate's constraints in the AIG. Concretely, for each clause $C \in T(\mathsf{s}, \mathsf{s'})$, the designer proves the existence of a gate g such that the polynomial encoding of C belongs to S_g, i.e.,

$$\gamma_\epsilon(C)(X) \in S_g,$$

which can be further reduced to subset checking in ZK.

The polynomial set S_g of each gate g consists of all polynomials induced by treating g as an AND gate and as a latch, respectively. We next describe the construction of these polynomials for each case.

3.2.4 Polynomials when the gate is an AND gate. Each two-input AND gate with input signals σ_a, σ_b and output signal σ_c is represented by the triple $\big(\epsilon(\sigma_a),\ \epsilon(\sigma_b),\ \epsilon(\sigma_c)\big)$. The logical constraint imposed by the gate requires

$$\sigma_c \leftrightarrow (\sigma_a \wedge \sigma_b).$$

Equivalently, the following three clauses must hold simultaneously:

$$\neg\sigma_c \vee \sigma_a \qquad \neg\sigma_c \vee \sigma_b \qquad \neg\sigma_a \vee \neg\sigma_b \vee \sigma_c. \tag{2}$$

Each clause corresponds to a polynomial whose roots are the encodings of its literals (with a negated literal $\neg\sigma$ encoded as $\alpha - \epsilon(\sigma)$. The three clauses thus induce the following polynomials:

$$\big(X - (\alpha - \epsilon(\sigma_c))\big)\big(X - \epsilon(\sigma_a)\big), \tag{3a}$$

$$\big(X - (\alpha - \epsilon(\sigma_c))\big)\big(X - \epsilon(\sigma_b)\big), \tag{3b}$$

$$\big(X - (\alpha - \epsilon(\sigma_a))\big)\big(X - (\alpha - \epsilon(\sigma_b))\big)\big(X - \epsilon(\sigma_c)\big). \tag{3c}$$

Since we also consider two consecutive steps with the transition relation, the following next-state counterparts of the polynomials in polynomials (3a)–(3c) must also be included in S_g:

$$\big(X - (\alpha - \epsilon^T(\sigma_c))\big)\big(X - \epsilon^T(\sigma_a)\big), \tag{4a}$$

$$\big(X - (\alpha - \epsilon^T(\sigma_c))\big)\big(X - \epsilon^T(\sigma_b)\big), \tag{4b}$$

$$\big(X - (\alpha - \epsilon^T(\sigma_a))\big)\big(X - (\alpha - \epsilon^T(\sigma_b))\big)\big(X - \epsilon^T(\sigma_c)\big), \tag{4c}$$

which can be further simplified by substituting $\epsilon^T(\sigma) = \beta + \epsilon(\sigma)$ for each signal $\sigma \in \{\sigma_a, \sigma_b, \sigma_c\}$, reducing each next-state polynomial to a shifted version of its current-state counterpart. The polynomial set S_g includes polynomials (3a)–(3c) and (4a)–(4c).

3.2.5 Polynomials when the gate is a LATCH. Each latch operates on an input signal σ_i and an output signal σ_o, propagating the value of σ_i in the current state to σ_o in the next state. Each latch is represented and committed by the triple $\big(\epsilon(\sigma_i),\ \epsilon(\sigma_o), \mathit{Init}(v)\big)$. The latch semantics impose the following constraint:

$$\big(\mathbf{s}.\sigma_i \leftrightarrow \mathbf{s}'.\sigma_o\big),$$

which ensures that the latch output in the next state matches the input signal in the current state, as constrained by the transition relation T. This biconditional is equivalent to the following two clauses:

$$(\sigma_i \vee \neg\sigma_o') \qquad \neg\sigma_i \vee \sigma_o'.$$

The polynomial set S_g therefore, also includes the following two polynomials:

$$\big(X - \epsilon(\sigma_i)\big)\big(X - (\alpha - \epsilon^T(\sigma_o))\big), \tag{5a}$$

$$\big(X - \epsilon^T(\sigma_o)\big)\big(X - (\alpha - \epsilon(\sigma_i))\big). \tag{5b}$$

3.2.6 Correctness verification of Init. The initial state predicate $\mathit{Init}(\mathsf{s})$ enforces that each latch $v \in L$ begins with its specified Boolean value $\mathit{Init}(v) \in \{0, 1\}$.

To check the correctness of $\mathit{Init}(\mathsf{s})$, we then require that for each clause $C_v \in \mathit{Init}(\mathsf{s})$ shown in Equation 6, there exists a LATCH $v \in L$, such that

$$\gamma_\epsilon(C_v)(X) = \begin{cases} X - \epsilon(\sigma_v), & \text{if } \mathit{init}(v) = 1, \\ X - \big(\alpha - \epsilon(\sigma_v)\big), & \text{if } \mathit{init}(v) = 0. \end{cases} \tag{6}$$

3.3 Proving Validity of Implication-Form Formulas in ZK

An inductive safety proof requires verifying three conditions over the committed CNF formulas Init, T, and Π. However, when these formulas are logically combined to express the inductive conditions, the resulting formulas are no longer in CNF and cannot be handled directly by existing zero-knowledge unsatisfiability backends. Specifically, two of the three conditions require negating and flattening an implication into a conjunction: the designer must prove the unsatisfiability of non-CNF formula $\mathit{Init}(\mathsf{s}) \wedge \neg\Pi(\mathsf{s})$ (initialization) and $\Pi(\mathsf{s}) \wedge T(\mathsf{s}, \mathsf{s'}) \wedge \neg\Pi(\mathsf{s'})$ (transition), where Init, T, and Π in CNF are privately held and only accessible via their commitments. However, we only have a backend for proving the unsatisfiability of CNF formulas.

A straightforward approach applies standard De Morgan transformations to obtain an equivalent CNF formula for $\neg\Pi$. However, this approach requires the designer to first commit to the De Morgan–transformed $\neg\Pi$, and then to prove its correctness, which can substantially increase time. For instance, a CNF invariant with k clauses of m literals can expand to $O(m^k)$ clauses, each with k literals in the worst case. Such a direct transformation causes an exponential time blow-up, making the encoding infeasible in practice (see the experimental results in Section 4.3 and Figure 5). To address this issue, ZSafe uses a probabilistic Tseytin–based ZK protocol, avoiding the formula size blow-up caused by direct De Morgan expansion.

In the rest of this section, we first prove that replacing $\neg\Pi$ in the implication with its Tseytin translation preserves soundness. We then introduce an intermediate protocol ZKUNSAT-IN^w for verifying the unsatisfiability of $\phi \wedge \neg\Pi$ with weakened soundness, arising from the absence of variable freshness checking. Finally, we present ZKUNSAT-IN, a probabilistic protocol that checks whether a committed CNF formula is correctly constructed from $\phi \wedge \neg\Pi$ for any committed CNF formula ϕ and Π in ZK.

Tseytin translation. Let $\Pi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$, where each $C_i = (l_{i,1} \vee l_{i,2} \vee \cdots \vee l_{i,m})$. By applying a Tseytin translation, we get a CNF $\widetilde{\Pi}$ that is *equisatisfiable* to DNF $\neg\Pi$ with an auxiliary variable set $A = \{z_1, \ldots, z_k\}$:

$$\widetilde{\Pi} = (z_1 \vee \cdots \vee z_k) \wedge \bigwedge_{i=1}^{k} \left[(l_{i,1} \vee \cdots \vee l_{i,m} \vee z_i) \wedge \bigwedge_{j=1}^{m} (\neg z_i \vee \neg l_{i,j}) \right] \tag{7}$$

Unlike the De Morgan approach, which produces a logically equivalent formula, the Tseytin translation yields an *equisatisfiable* formula of size $O(km)$, which is bilinear in the size of the original formula Π.

We next prove that such a translation can be used in our setting without compromising soundness.

Lemma 3.1. *Let $\Pi = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ over variables X, and let $\widetilde{\Pi}$ be $\neg\Pi$'s Tseytin obtained by introducing a set of fresh auxiliary variables Z (so $Z \cap X = \varnothing$). Then, for every model $\mathcal{M}$ assigning truth values to variables in X,*

$$\mathcal{M} \models \neg\Pi \quad \Longleftrightarrow \quad \exists\, \mathcal{M}_Z \text{ such that } \mathcal{M} \cup \mathcal{M}_Z \models \widetilde{\Pi},$$

where $\mathcal{M}_Z$ assigns truth values to variables in Z, and $\mathcal{M} \cup \mathcal{M}_Z$ denotes the combined assignment.

Proof. ($\Rightarrow$) Suppose $\mathcal{M} \models \neg\Pi$. We construct $\mathcal{M}_Z$ on the structure of Π. For each clause C_i of Π, Tseytin introduces a fresh variable z_i with clauses enforcing $z_i \leftrightarrow \neg C_i$. Define $\mathcal{M}_Z(z_i)$ the Boolean evaluation of z_i under $\mathcal{M}$. Since $\mathcal{M}$ satisfies $\neg\Pi$, at least one C_i satisfies $\mathcal{M}(\neg C_i) = 1$, therefore $\mathcal{M}_Z(z_i) = 1$ and also $\mathcal{M}_Z(z_1 \vee z_2 \vee \cdots \vee z_k) = 1$. For each $i \in \{1, \cdots, k\}$, if $\mathcal{M}_Z(z_i) = 1$, then $\mathcal{M}(\neg C_i) = 1$, which implies $\ell_{ij} = 0$ and $(\neg z_i \vee \neg \ell_{ij})$ for each $j \in \{1, \cdots, m\}$. If $\mathcal{M}_Z(z_i) = 0$, then $\mathcal{M}(\neg C_i) = 0$, which means at least one j such that $\ell_{ij} = 1$ and therefore $(\neg z_i \vee \neg \ell_{ij})$ for each $j \in \{1, \cdots, m\}$. In both cases, all clauses of $\widetilde{\Pi}$ are satisfied, so $\mathcal{M} \cup \mathcal{M}_Z \models \widetilde{\Pi}$.

($\Leftarrow$) Conversely, suppose $\mathcal{M} \cup \mathcal{M}_Z \models \widetilde{\Pi}$ for some $\mathcal{M}_Z$. By construction of $\widetilde{\Pi}$, the clause $(z_1 \vee \cdots \vee z_k)$ must be satisfied, so there exists some i such that $\mathcal{M}_Z(z_i) = 1$. The Tseytin clauses enforce $z_i \leftrightarrow \neg C_i$; since $\mathcal{M}_Z(z_i) = 1$, we have $\mathcal{M}(\neg C_i) = 1$, i.e., $\mathcal{M} \not\models C_i$. Since $\Pi = C_1 \wedge \cdots \wedge C_k$ requires all clauses to hold and C_i is violated, it follows that $\mathcal{M} \models \neg\Pi$. □

Therefore, the Tseytin translation is a conservative equisatisfiable extension: every model of $\neg\Pi$ extends to a model of $\widetilde{\Pi}$, and every model of $\widetilde{\Pi}$ restricted to X satisfies $\neg\Pi$.

Theorem 3.2. *Let $\widetilde{\Pi}$ be the CNF formula translated from DNF $\neg\Pi$ via the Tseytin transformation. For any propositional formula ϕ, if $\phi \wedge \widetilde{\Pi}$ is unsatisfiable, then $\phi \wedge \neg\Pi$ is also unsatisfiable.*

Proof. We proceed by contradiction. Assume, for the sake of contradiction, that $\phi \wedge \neg \Pi$ is satisfiable. Then there exists at least one model (a truth assignment to all variables) $\mathcal{M}$ that satisfies both ϕ and $\neg \Pi$. This means:

$$\exists \mathcal{M} : \mathcal{M} \models \phi \wedge \neg \Pi$$

Therefore, we can find a model $\mathcal{M}$, such that $\mathcal{M} \models \phi$ and $\mathcal{M} \models \neg \Pi$. As the Tseytin transformation preserves satisfiability, there must exist a model $\mathcal{M}'$ that satisfies $\widetilde{\Pi}$. Besides, according to Lemma 3.1, this model $\mathcal{M}'$ is an extension of $\mathcal{M}$ where the assignments for the original variables in $\mathcal{M}$ (including those in ϕ) are preserved. Thus,

$$\exists \mathcal{M}' : \mathcal{M}' \models \phi \wedge \widetilde{\Pi}$$

This directly implies that $\phi \wedge \widetilde{\Pi}$ is satisfiable, which contradicts our initial premise. □

By Theorem 3.2, the designer can prove the validity of $\phi \Rightarrow \Pi$ by showing that $\phi \wedge \widetilde{\Pi}$ is UNSAT, where $\widetilde{\Pi}$ is the CNF of the Tseytin translation of $\neg \Pi$.

In what follows, we present an *intermediate* protocol, ZKUNSAT-IN^w, with weakened soundness for checking $\phi \Rightarrow \Pi$ based on the Tseytin translation. We then identify the reasons for the soundness gap and introduce a ZK protocol, obtained by modifying the ZKUNSAT-IN^w, that enables the verifier to ensure $\phi \Rightarrow \Pi$ with high confidence.

ZKUNSAT-IN^w for $\phi \wedge \neg \Pi$. In this protocol, a designer prepares and commits a private set of clauses $C_{\widetilde{\Pi}} = \{C_i^{\Pi}, D_{ij}^{\Pi}, C_z\}$, as a set of private polynomials using encoding in Section 2.4, as well as a private set of encoding $\{e_{i,1}, \cdots, e_{i,m}\}$ along with public encoding $\{\epsilon(z_1) \cdots, \epsilon(z_k)\} \in \mathbb{F}^k$ that has not been used before, and then prove in ZK the following constraints hold:

$$(X - e_{i,1}) \times \cdots \times (X - e_{i,m}) = \gamma_\epsilon(C_i)(X) \quad \text{for each } C_i \in C_{\Pi}, \tag{8a}$$

$$\gamma_\epsilon(C_i)(X)(X - \epsilon(z_i)) = \gamma_\epsilon(C_i^{\Pi})(X), \tag{8b}$$

$$(X - (\alpha - e_{i,j}))(X - \epsilon(\neg z_i))) = \gamma_\epsilon(D_{ij}^{\Pi})(X), \tag{8c}$$

$$(X - \epsilon(z_1)) \times \cdots \times ((X - \epsilon(z_k))) = \gamma_\epsilon(C_z)(X). \tag{8d}$$

The designer then commits a private CNF h together, proving the unsatisfiability of h and $C_h \subseteq C_{\widetilde{\Pi}} \cup C_\phi$ via ZKUNSAT and subset checking in ZK. The statement that the designer proves establishes only a relaxed condition that $\phi \wedge \neg \Pi$ meets. We next explain the relaxation.

First, the verifier can not enforce that every clause of Π is used when constructing $\widetilde{\Pi}$, nor that the tuples $\{e_{i,1}, \ldots, e_{i,m}\}$ are drawn from distinct clauses of Π; they could all come from a single clause, and the subset-inclusion check would not detect this because these tuples are committed as private values. Second, the verifier checks only a subset of the clauses in $\phi \wedge \widetilde{\Pi}$ that forms an unsatisfiable core, rather than the entire $\phi \wedge \widetilde{\Pi}$. We now show that this relaxation does not affect soundness.

Lemma 3.3. *Suppose $\widetilde{\Pi}$ and $\widetilde{\Pi}_{sub}$ are CNF formulas with $C_{\widetilde{\Pi}_{sub}} \subseteq C_{\widetilde{\Pi}}$. If $\phi \wedge \widetilde{\Pi}_{sub}$ is unsatisfiable, then $\phi \wedge \widetilde{\Pi}$ is also unsatisfiable.*

Proof. Assume for contradiction that $\phi \wedge \widetilde{\Pi}$ is satisfiable. Then there exists a model $\mathcal{M}$ such that $\mathcal{M} \models \phi$ and $\mathcal{M} \models \widetilde{\Pi}$. Since every clause of $\widetilde{\Pi}_{\text{sub}}$ appears in $\widetilde{\Pi}$, it follows that $\mathcal{M} \models \widetilde{\Pi}_{\text{sub}}$. Hence $\mathcal{M} \models \phi \wedge \widetilde{\Pi}_{\text{sub}}$, contradicting the unsatisfiability of $\phi \wedge \widetilde{\Pi}_{\text{sub}}$. Therefore, $\phi \wedge \widetilde{\Pi}$ is unsatisfiable. □

Theorem 3.4. *Let Π be a CNF, and let $\widetilde{\Pi}$ be a CNF with clause set $C_{\widetilde{\Pi}}$ that satisfies the conditions in Equation 8 for Π. Assume $\{z_1, \ldots, z_m\}$ are fresh auxiliary variables whose encodings do not occur in ϕ or Π. If $\phi \wedge \widetilde{\Pi}$ is unsatisfiable, then $\phi \wedge \neg \Pi$ is unsatisfiable.*

Proof. The proof directly follows Lemma 3.3 and CNF formula 7 from Tseytin translation for Π. □

Rather than proving the unsatisfiability of $\phi \wedge \widetilde{\Pi}$ directly in zero-knowledge, ZSafe only requires the prover to demonstrate the existence of a CNF formula h with $C_h \subseteq C_\phi \cup C_{\widetilde{\Pi}}$ such that $\phi \wedge h$ is unsatisfiable. This is motivated by the fact that $\phi \wedge \widetilde{\Pi}$ can be very large, whereas its unsat core is typically much smaller. By proving the unsatisfiability of the unsat core rather than the full formula, ZSafe makes privacy-preserving verification practical.

The following lemma shows that this weakened condition still preserves soundness.

Lemma 3.5. *Let ϕ be a CNF formula. Then ϕ is unsatisfiable if and only if there exists a subset $C' \subseteq C_\phi$ such that the formula $\bigwedge_{C \in C'} C$ is unsatisfiable.*

Proof. **(⇐)** Suppose there exists a subset $C' \subseteq C_\phi$ such that $\bigwedge_{C \in C'} C$ is unsatisfiable. For contradiction, assume that ϕ is satisfiable. Then there exists an assignment σ such that $\sigma \models \phi$, i.e., σ satisfies every clause in C_ϕ. Since $C' \subseteq C_\phi$, it follows that σ also satisfies every clause in C'. But this contradicts the assumption that $\bigwedge_{C \in C'} C$ is unsatisfiable. Hence, ϕ must also be unsatisfiable. **(⇒)** Let C' be C. □

Finally, the verifier cannot enforce the freshness of the auxiliary variables. The private set $\{\epsilon(z_1), \ldots, \epsilon(z_k)\}$, if committed by the designer, may reuse encodings already used in Π or ϕ. The correctness of $\widetilde{\Pi}$ depends on these variables being fresh: if a Tseytin variable collides with a variable occurring in ϕ or Π, satisfiability can change. Thus, a malicious designer could select non-fresh auxiliaries, remain undetected, and convince the verifier that $\phi \wedge \widetilde{\Pi}$ UNSAT even when it is not. We propose a modification of ZKUNSAT-IN^w that addresses this issue and offers the desired soundness.

ZKUNSAT-IN for $\phi \wedge \neg\Pi$. In this protocol, with Π and ϕ committed, the *verifier* first samples $r_1, \ldots, r_m$ uniformly at random and sends them to the designer as new variable encodings. The designer then returns committed clause set $C_{\widetilde{\Pi}}$ and the encodings $\{e_{i,1}, \ldots, e_{i,m}\}$ (as above). The verifier proceeds with the remaining ZKUNSAT-IN^w checks on these inputs and the agreed pool of fresh variables.

Theorem 3.6. *Let K be the number of literal* occurrences *already used in ϕ and Π (counting multiplicity), and let $N := |\mathcal{E}|$ be the size of the encoding pool $\mathcal{E} = \{u \in \mathbb{F} : u$ has a variable preimage under $\epsilon\}$. Then the probability that the designer can cheat in ZKUNSAT-IN is at most $\frac{KM}{N-M+1}$. When $M \ll N$, this probability is bounded by $1 - \exp\left(-\frac{KM}{N}\right)$. Let M be the number of new auxiliary encodings the verifier must choose (without replacement) from $\mathcal{E}$.*

Proof. The designer can cheat only if at least one of the M sampled encodings collides with the K encodings already used in ϕ or Π. The probability of *no* collision is

$$p_{\text{fresh}} = \frac{\binom{N-K}{M}}{\binom{N}{M}} = \prod_{i=0}^{M-1} \frac{N-K-i}{N-i} \;\geq\; \left(1 - \frac{K}{N-M+1}\right)^M.$$

Therefore the cheating probability satisfies

$$p_{\text{cheat}} = 1 - p_{\text{fresh}} \;\leq\; 1 - \left(1 - \frac{K}{N-M+1}\right)^M \;\leq\; \frac{KM}{N-M+1},$$

where the last inequality follows from Bernoulli's inequality.

□

Remark: When $K, M \ll N$, we can further approximate the probability and obtain $p_{\text{cheat}} \approx 1 - \exp\left(-\frac{KM}{N}\right)$. We show in Figure 10 that in the practical implementation setting, this probability is negligible.

3.4 Shared-Structure Unsatisfiability and Clause Consistency

The protocol above verifies each inductive proof obligation separately. A sound inductive argument must also bind the invariant clauses that recur across these obligations and relate their current-state and next-state encodings. We address these two requirements separately.

Shared structure across proof obligations. The initialization, transition, and safety-implication checks all depend on the same private invariant. We refer to the invariant clauses reused across these obligations as the shared structure. ZSafe commits $\Pi(\mathtt{s})$ once and uses the same commitment in every proof obligation containing it. Without this binding, a malicious designer could use a different invariant in each obligation, causing the verifier to accept an invalid safety proof.

However, since $\Pi(\mathtt{s})$ and $\Pi(\mathtt{s'})$ are committed independently, a malicious designer could breach the protocol, such as substituting two distinct invariants Π_1 and Π_2 and proving the unsatisfiability of

$$\mathit{Init}(\mathtt{s}) \wedge \neg\Pi_1(\mathtt{s}), \qquad \Pi_1(\mathtt{s}) \wedge T(\mathtt{s}, \mathtt{s'}) \wedge \neg\Pi_2(\mathtt{s'})$$

for irrelevant Π_1 and Π_2, without the verifier detecting the inconsistency. ZSafe therefore requires the designer to additionally prove that the committed CNFs for $\Pi(\mathtt{s})$ and $\Pi(\mathtt{s'})$ are consistent, i.e., they encode the same invariant over their respective state variables in ZK. Next, we explain how ZSafe implements this.

Recall that the encoding of a CNF formula is the set of encodings of its clauses. ZSafe forces the designer to commit the following clause sets (equivalently, the corresponding CNF formulas):

$$C_{\mathit{Init}}, \quad C_T, \quad C_\Pi, \quad C_{\Pi'}, \quad C_{\mathit{Bad}},$$

where $C_\Pi = \{C_i\}_{i=1}^k$ and $C_{\Pi'} = \{C'_j\}_{j=1}^k$ are the CNF clause sets of $\Pi(\mathtt{s})$ and $\Pi(\mathtt{s'})$, respectively. Initialization and safety implication share the clause set $C_\Pi = \{C_i\}_{i=1}^k$. The sets C_{Init} and C_T are encodings of the design and are checked in ZK for correctness as in Section 3.2.

Clause consistency check between $\Pi(\mathtt{s})$ and $\Pi(\mathtt{s'})$. We require the designer to show in ZK that

$$\forall\, C \in C_\Pi\ \exists\, C' \in C_{\Pi'} \text{ such that} \quad \gamma_\epsilon(C)(X+\beta) = \gamma_\epsilon(C')(X),$$

$$\forall\, C' \in C_{\Pi'}\ \exists\, C \in C_\Pi \text{ such that} \quad \gamma_\epsilon(C)(X+\beta) = \gamma_\epsilon(C')(X),$$

where $\beta \in \mathbb{F}$ is the fixed shift that maps each current-state literal to its next-state copy (i.e., $\epsilon^T(\ell) = \epsilon(\ell') = \epsilon(\ell) + \beta$. This proves that every clause of $\Pi(\mathtt{s})$ has a correctly renamed counterpart in $\Pi(\mathtt{s'})$. [1]

3.5 Optimization

3.5.1 Balanced Tseytin Translation. Recall that the equisatisfiable CNF $\widetilde{\Pi}$ obtained from Equation 7 may contain clauses of width up to $\max(k, m+1, 2)$, which becomes a major bottleneck in unsatisfiability check as its cost scales with the maximum clause width. To address this, we propose *Balanced Tseytin Translation* to enforce a uniform bound ω on clause width.

[1] In general, the designer can prove there exists a bijection π between the clause sets of $\Pi(\mathtt{s})$ and $\Pi(\mathtt{s'})$, $\gamma_\epsilon(C)(X+\beta) = \gamma_\epsilon(\pi(C))(X)$ for every $C \in C_\Pi$.The existence of such a permutation can be proved in ZK using standard permutation arguments. We adopt this bijection-based approach in our implementation. We omit the protocol details and focus on the principal idea here.

For any clause $(x_1 \vee \cdots \vee x_t)$ in Equation 7, we replace it by a set of Tseytin clauses that introduce auxiliary variables

$$\begin{aligned}\mathrm{B}_\omega(x_1,\ldots,x_t) \;=\; & \bigwedge_{i=1}^{\lceil t/\omega \rceil} \left[\left(\bigvee_{j=(i-1)\omega+1}^{\min(i\omega,t)} x_j \;\vee\; \neg z_i^{(1)} \right) \wedge \bigwedge_{j=(i-1)\omega+1}^{\min(i\omega,t)} (\neg x_j \vee z_i^{(1)}) \right] \\ & \wedge \;\cdots \\ & \wedge \bigwedge_{i=1}^{\lceil t/\omega^d \rceil} \left[\left(\bigvee_{j=(i-1)\omega+1}^{\min(i\omega,\lceil t/\omega^{d-1} \rceil)} z_j^{(d-1)} \;\vee\; \neg z_i^{(d)} \right) \wedge \bigwedge_{j=(i-1)\omega+1}^{\min(i\omega,\lceil t/\omega^{d-1} \rceil)} (\neg z_j^{(d-1)} \vee z_i^{(d)}) \right] \\ & \wedge \; z_1^{(d)}.\end{aligned}$$

where all $z_i^{(\ell)}$ are fresh auxiliary variables, and d is the smallest integer such that $\lceil t/\omega^d \rceil = 1$.

Applying this transformation to both $A = (z_1 \vee \cdots \vee z_k)$ and each clause $C_i = (l_{i,1} \vee \cdots \vee l_{i,m} \vee z_i)$ in Equation 7, we obtain

$$\widetilde{\Pi} = \mathrm{B}_\omega(z_1,\ldots,z_k) \;\wedge\; \bigwedge_{i=1}^{k} \left[\mathrm{B}_\omega(l_{i,1},\ldots,l_{i,m},z_i) \;\wedge\; \bigwedge_{j=1}^{m} (\neg z_i \vee \neg l_{i,j}) \right],$$

By construction, the resulting CNF $\widetilde{\Pi}$ remains equisatisfiable to $\neg\Pi$, and every introduced clause has width at most $\omega + 1$. We further demonstrate in Section 4.7 that this optimization leads to substantial reductions in the overall proving time.

3.5.2 Strengthened Invariant. Recall that the invariant Π is privately computed by the designer. To prove design safety to the verifier, the designer must demonstrate the validity of these three implication checks in zero knowledge, each of which incurs substantial cryptographic overhead.

Therefore, we strengthen the invariant by conjoining the negation of the bad-state predicate into it, such that

$$\widehat{\Pi}(\mathsf{s}) = \Pi(\mathsf{s}) \wedge \neg \mathit{Bad}(\mathsf{s}). \tag{9}$$

Since $\widehat{\Pi}(\mathsf{s}) \implies \neg \mathit{Bad}(\mathsf{s})$ holds by construction of Equation 9, we further show that, by Lemma 3.7, the reduction is sound, i.e., validating the strengthened initialization and strengthened transition suffices to establish the safety property.

$$\mathit{Init}(\mathsf{s}) \implies \widehat{\Pi}(\mathsf{s}) \qquad \text{(Strengthened Initialization)}$$

$$\widehat{\Pi}(\mathsf{s}) \wedge T(\mathsf{s},\mathsf{s}') \implies \widehat{\Pi}(\mathsf{s}') \qquad \text{(Strengthened Transition)}$$

Lemma 3.7. *Let $\widehat{\Pi}(\mathsf{s}) = \Pi(\mathsf{s}) \wedge \neg \mathit{Bad}(\mathsf{s})$. If strengthened initialization and strengthened transition hold, then the system satisfies the safety property.*

Proof. By definition of strengthened initialization and strengthened transition, all reachable states should satisfy $\widehat{\Pi}$. By definition of $\widehat{\Pi}$, we have $\widehat{\Pi}(\mathsf{s}) \implies \neg \mathit{Bad}(\mathsf{s})$ for every state s. Therefore, no reachable state satisfies *Bad*, and thus the safety property holds. □

The designer proves the correctness of $\widehat{\Pi} = \Pi \wedge \neg \mathit{Bad}$ using the same protocol described in Section 3.2. The strengthened implication checks are verified using the same procedure described in Section 3.3.

4 Implementation and Evaluation

We implement ZSafe, which can be directly integrated with existing open-source hardware verification frameworks. Specifically, the hardware design is first synthesized into an AIG representation on the designer side using Yosys [58]. Meanwhile, the verifier specifies the safety property in temporal logic and converts it into a property AIG using a monitor circuit compiler [20]. Safety verification is carried out by synchronizing the property AIG with the design, where the property signals are routed from the design AIG as the input to the property AIG [1].

4.1 Setup

All experiments were conducted on an AWS `r5b.16xlarge` instance equipped with 64 vCPUs and 512 GB of memory. High-memory instances were provisioned to meet the substantial RAM requirements of the largest benchmark cases. The designer and verifier supply the design and property, respectively, as two parties.

We implement ZSafe's ZK backend with EMP-TOOLKIT[57] and derive the framework based on ZKUNSAT[39]. Resolution proofs for unsatisfiability are generated by PICOSAT[10], and their traces are optimally reordered with TRACETRACK[9]. The safety invariant is produced by RIC3 [54] with its default 10-thread IC3 engine.

4.2 Benchmark

To assemble a diverse and challenging testbed including complex designs and hard instances, we repurpose the 2024 Hardware Model Checking Competition (HWMCC'24) suite [3] as our benchmark set, as shown in the x-axis of Figure 3, including peripheral controllers (`qspiflash_*`), processors (`VexRiscv_*`, `picorv32_*`, etc.), arithmetic circuits (`cal*`), etc.

The property and design are prepared as follows. Since each HWMCC'24 benchmark contains an embedded `bad` output encoding the bad-state predicate *Bad*, we use the transitive fan-in cone of this output to construct a synthetic public/private partition. Specifically, we traverse the fan-in cone with a budget equal to 10% of the benchmark's total AND/LATCH gate count to form the public property AIG and treat all unselected gates as the private design AIG. As shown in Figure 3, the resulting property AIGs contain 23 to 5,636 AND/LATCH gates, while the complete benchmark AIGs contain 288 to 308,022 gates. This property-size range is sufficient for evaluating nontrivial safety properties, and the use of property AIGs is natural in this setting, as alternation-free formulas admit linear-size circuit translations [20].

Note. While the public/private partition is synthetic, it does not affect the scalability conclusions of ZSafe presented in Section 4.3. As a fraction of the total circuit, the public properties' share is negligible because our protocol's cost is dominated by ZK validation of the logical implications, which is agnostic to this fraction. Even if the benchmark is more conservative and set 100% of the partition to be private, this would only increase overhead by up to 5%.

4.3 End-to-End Evaluation

We begin by presenting the end-to-end evaluation with optimization from the strengthened invariant (in Section 3.5.2), followed by a discussion of the Balanced Tseytin Translation's effects (in Section 3.5.1).[2]

Figure 4 shows the runtime of each instance. The red curve depicts the cumulative proportion of instances completed before time t, while the blue bars indicate the fraction of instances finishing

[2]Inductive proof checking primarily contributes to the total runtime, others, including design encoding, property encoding, and invariant encoding, also incur non-negligible costs. We omit the time cost of invariant generation and property compilation, as they can be performed independently of the ZSafe framework.

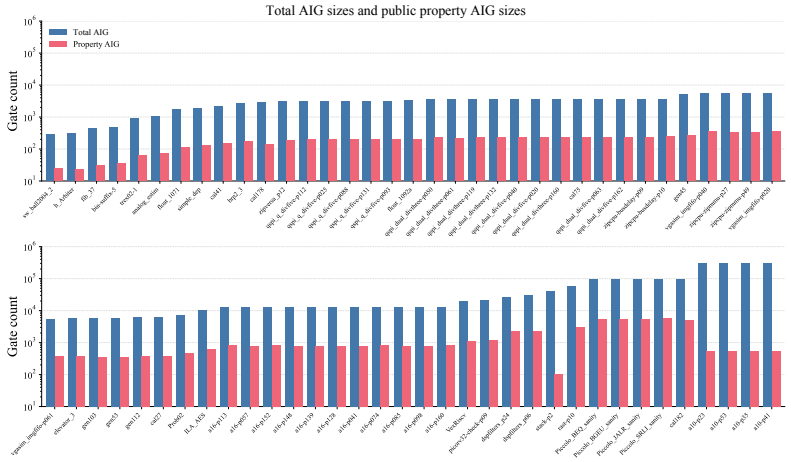


Fig. 3. Total AIG sizes and public property AIG sizes across the adapted HWMCC'24 benchmarks. Paired bars show the total AIG and its public property component. Total AIGs range from 288 to 308,022 AND/LATCH gates, while public property AIGs range from 23 to 5,636 gates.

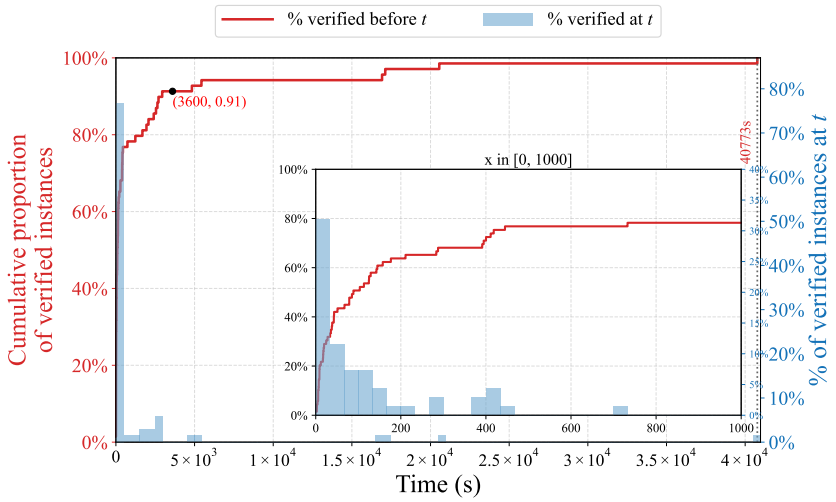


Fig. 4. ZSafe's performance against HMWCC benchmark. The red curve depicts the cumulative proportion of instances completed before time t, while the blue bars indicate the fraction of instances finishing at time t. More than half finish within 200 seconds, and 90% within an hour.

at time t. ZSafe is capable of completing proofs within hours, where over half of the instances complete the check within 200 seconds, and over 90% finish within an hour. The hardest case, gen112, required 40 thousand seconds (≈11.3 hours), as highlighted by the vertical marker.

In Figure 5, we illustrate the time distribution using a stacked bar chart in percentage form, with the total runtime annotated above each bar. Instances are ordered in ascending total runtime, with the rightmost instance, gen112, taking 40772 seconds to complete, and the smallest instance h_arbiter takes only 2.2 seconds. Instance names are color-coded by size, with darker shades indicating larger ones, as shown in the size bar on the right y-axis side. The design size corresponds

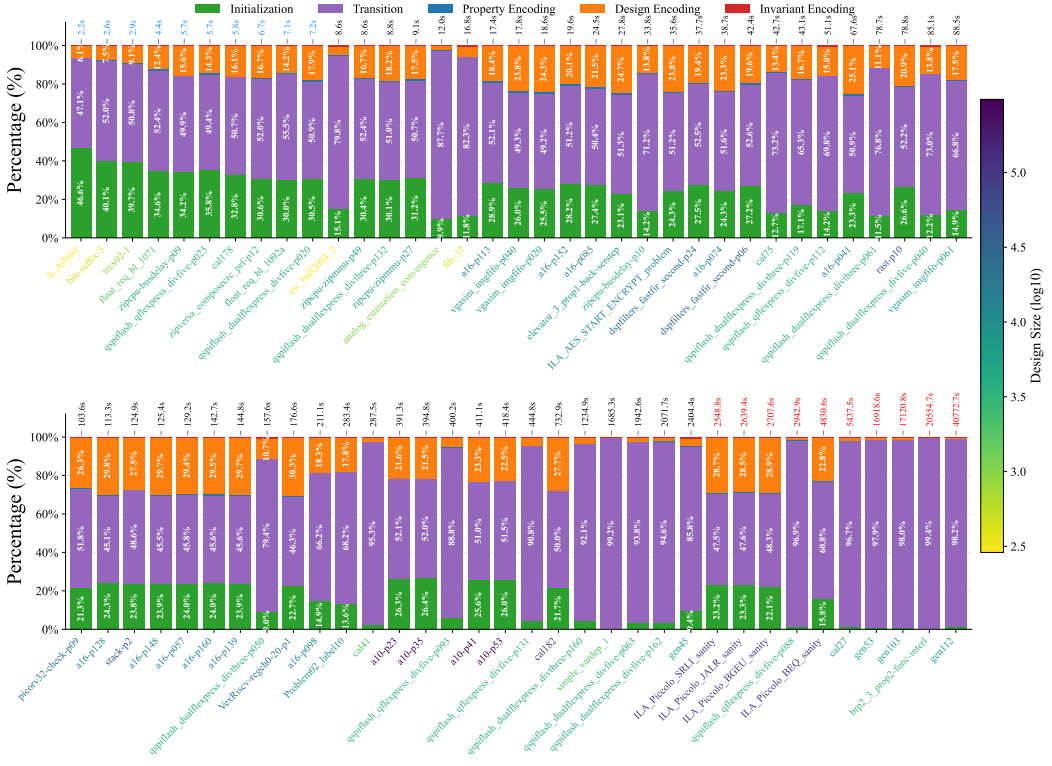


Fig. 5. The percentage breakdown of Initialization, Transition, Property & Invariant Encoding, and Design Encoding times for each design instance. End-to-end execution times are shown above each bar, sorted in ascending order. X-axis labels are colored according to the logarithm of design size using the colormap shown at right, where darker colors correspond to larger designs.

to the total number of AND and LATCH gates. The largest instances we evaluate are series of `a10_p*` with 0.3 million gates, and the smallest is `sw_ball2004_2`, which contains 288 gates. The time cost does not scale proportionally with instance size; for example, `gen112` with the longest proof time contains only around 6,000 gates because the resolution proof length and maximal clause width are other factors influencing the cost. `gen112` has a resolution proof length of 0.2 million in the transition check with a maximal clause width of 1257, which causes more than 98% of the time to be spent in the transition check.

Initialization checking and design encoding also contribute significantly to the runtime, each accounting for up to approximately 25% in about half of the cases. Initialization is generally considered less complex than transition, and its cost does not exceed that of the transition. The time for property encoding and invariant encoding is negligible compared with others because these steps mostly amount to committing to CNF formula and performing lightweight consistency checks. Furthermore, the invariants contain far fewer clauses compared with the instance size. For example, the invariant of the largest instance `a10_p23` only has three clauses.

4.4 Design Encoding Time

Design encoding time refers to the time for the arithmetization and consistency-checking of the design encoding, as discussed in Section 3.2. Figure 6a plots the maximum clause width against the

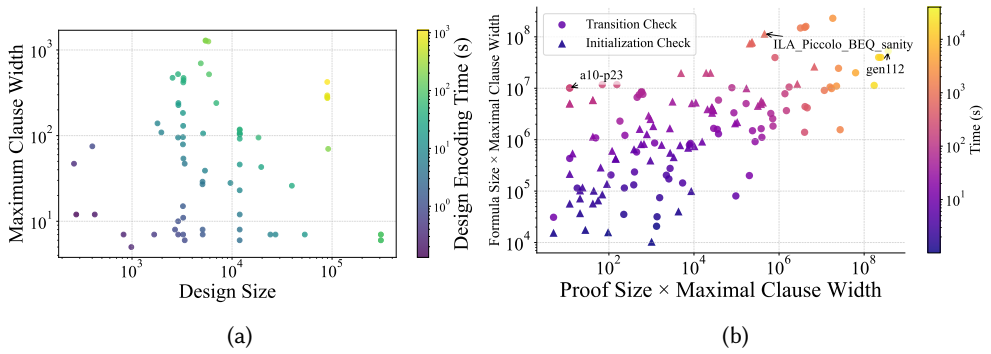


Fig. 6. (a) Scatter plot of design size (total AND+LATCH gates) versus maximal clause width, with color indicating design encoding time. Encoding time increases with design size and grows further with maximal clause width; (b) Scatter plot where circles denote transition checks and triangles denote initialization checks. Transition checks are generally more expensive than initialization checks, with instances influenced by both proof length and formula size.

design size, with color indicating the design encoding time in seconds. `ILA_Piccolo_BEQ_sanity` takes the longest time (1000s) in experiments, which is derived from the abstraction of the RISC-V and contains more than 90,000 gates. `sw_ball2004_2` and `h_Arbiter` take less than a second.

The time cost scales with two factors: the design size and the maximal clause width. Though `ILA_Piccolo_BEQ_sanity` is not the largest instance, it exhibits a maximal clause width of 423, which determines the degree of the polynomial used to encode the design constraints. A larger clause width requires more polynomial coefficients to be encoded, thereby incurring a higher time cost. Moreover, each AND gate may need to be encoded across two consecutive steps [3], which is more costly than LATCH encoding. Thus, designs of the same size with a higher proportion of AND gates tend to require more time.

4.5 Safety & Invariant Encoding

Verifying safety and invariant encoding is considerably easier than encoding the design, since the AIG-to-CNF translation is publicly performed. The encoding time scales linearly with the product of the maximum clause width and the number of clauses in Figure 7, with the longest case in our experiments taking only about 20 seconds.

The inductive invariant, provided by the prover as a private input, is encoded into CNF and further transformed into equisatisfiable CNF (discussed in Section 3.3) whose cost scales with the product of the invariant size and the maximal clause width, as shown in Figure 7. The invariant is much smaller than the design; therefore, the largest encoding in our experiments required only about 35 seconds.

4.6 Unsatisfiability Checking

The runtime of verifying the unsatisfiability of both the initialization and transition formulas grows with three key factors: the maximal clause width, the resolution proof size, and the size of the original input formula. Figure 6b illustrates this relationship, where circles represent the transition formula check and triangles represent the initialization formula check. The x-axis shows the product of proof size and maximal clause width, while the y-axis shows the product of input formula size and maximal clause width. The color scale encodes the proving time.

[3]The AND gate must be unrolled in two cases: (i) when the property includes the invariant constraints [12], and (ii) when the invariant involves the internal signal [18].

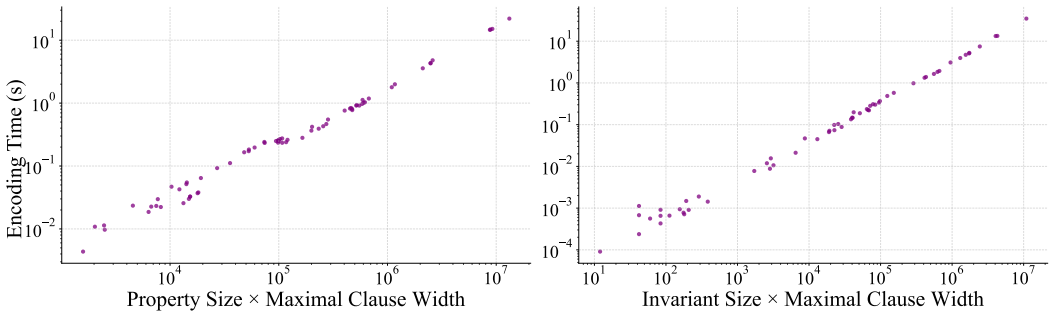


Fig. 7. Scatter plot of encoding time versus the product of formula size and maximal clause width, for both safety property and invariant encoding. Encoding time grows consistently with this product.

In general, the initialization check tends to be less time-consuming than the transition check. Because the transition check usually involves larger formulas and longer proofs, which is consistent with the intuition that proving the invariant is preserved under transitions is inherently more difficult than proving it holds initially.

Instances with the largest transition check cost do not necessarily have the largest initialization check cost. In our experiments, the design `gen112` exhibits the longest transition check runtime, where the resolution proof length reaches nearly 300,000 while the original formula contains only about 3,000 clauses. In this case, the proving cost is driven primarily by proof complexity rather than formula size. In contrast, the most time-consuming initialization check occurs in `ILA_Piccolo_BEQ_sanity`. This instance is not the hardest in terms of resolution proof length; instead, its high runtime is dominated by the large size of the original formula. Since the induction proof in ZKP involves both the resolution proof and the subset checking overhead. Another similar case is `a10p23`, where the resolution proof consists of only 2 clauses, yet it still incurs a relatively large proving cost due to the size of the input formula, which is around 1,683,000 clauses.

4.7 Optimization.

4.7.1 Balanced Tseytin Translation. The maximal clause width is a key factor in both instance encoding and induction proof, especially in instance encoding, as the design and property sizes are fixed; reducing clause width directly lowers encoding time. As shown in Figure 5, design encoding time accounts for up to 25% of the total runtime in nearly half of the designs. However, the maximal clause width is determined by the input formula and its resolution proof. Because the design and property are encoded as AIGs, their clauses have a maximal width of 3, which is relatively small. Thus, the maximal clause width is mainly contributed to by the invariant and the resolution proof.

Without structural analysis of the formula, the maximal clause of the resolution proof remains difficult to predict. It can, however, be influenced by the input formula, particularly the invariant. We observe that invariants with large maximal widths tend to result in resolution proofs with correspondingly larger maximal clause widths. This is intuitive, since unsatisfiability is ultimately derived from the invariant, which often serves as the key resolver in the resolution proof tree.

Figure 8 summarizes the impact of the Balanced Tseytin Transformation on maximal clause width and the estimated time cost as the maximal invariant-clause width is varied. The left y-axis plots the estimated time cost for the initialization and transition checks, computed as (input formula size + proof size)× (maximal clause width), under different invariant-width caps w. This metric is intended to approximate the runtime discussed in Section 4.3. The right y-axis reports the resulting maximal clause width. Colors indicate the Balanced Tseytin width limits applied to the invariant ($\omega \in$

$\{5, 25, 50, 100, 200, 400\}$); Circular markers denote the estimated time cost, and triangular markers denote the maximal clause width. Design names are ordered as in Figure 5.

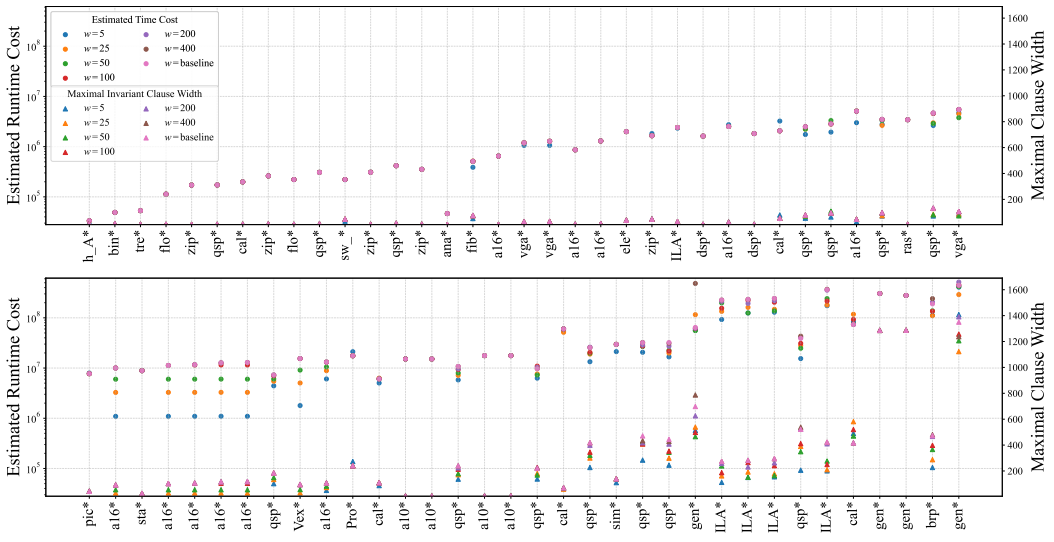


Fig. 8. The left y-axis shows the estimated time cost with different ω set for balanced Tseytin transformation. This metric is intended to approximate the runtime discussed in Section 4.3 . The right y-axis reports the maximal clause width. Different colors correspond to balanced Tseytin transformation with width limits applied to the invariant ($\omega \in \{5, 25, 50, 100, 200, 400\}$), with circular markers denoting estimated proof cost and triangular markers denoting maximal clause width. The design name is sorted in the same order as in Figure 5

.

Overall, the optimization is effective; the results show that in 56/69 cases the transformation had a non-negative effect (either reducing the final maximal clause width or leaving it unchanged), in 7/69 cases it was unstable (some parameter settings reduced the width while others increased it), and in the remaining 6/69 cases it was non-positive (either unchanged or increased). Reducing the maximal clause width does not necessarily improve resolution proof time, since decomposing a clause also increases the total number of clauses. To assess the net effect, we estimated the time cost using an approximation: the product of the maximal clause width and the combined size of the proof and the input formula. A clear majority (51/69, 74%) benefits or is unaffected, with only a small minority (7/69, 10%) showing non-positive impact. The remaining 11 cases show instability rather than systematic degradation. This result indicates a generally positive trend in resolution time cost.

Moreover, we conducted an end-to-end evaluation against the baseline in Figure 11 with $\omega = 100$. This setting primarily targets designs with large invariants. By choosing a moderate width, ZSafe improves efficiency in handling complex cases while mitigating the use of fresh variables. Across 69 instances, 12 improved, 1 degraded, and 56 were unchanged, which is consistent with our approximate estimation: designs with simple invariants do not require optimization, whereas harder instances benefit from it with significant improvements. The design `brp2_3_prop2-func-interl` achieved the largest runtime reduction from 20000s to 11000s (45%).

4.7.2 Comparison with Direct De Morgan Expansion. We further compare ZSafe's probabilistic Tseytin-based implication protocol with a natural baseline that expands $\neg II$ using direct De Morgan transformation. The De Morgan baseline does not introduce auxiliary variables, but it can grow

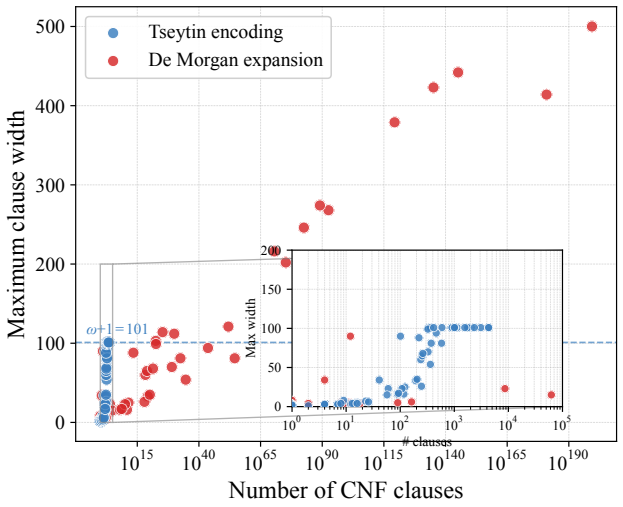


Fig. 9. Clause-count and clause-width comparison between direct De Morgan expansion and ZSafe's probabilistic Tseytin-based encoding. Each point is a benchmark from HWMCC'24. De Morgan expansion avoids auxiliary variables but grows exponentially, while the balanced Tseytin encoding with $\omega = 100$ keeps introduced clauses within the $\omega + 1 = 101$ width bound.

exponentially in the number of invariant clauses. As shown in Figure 9, this exponential blow-up is substantial in practice: among the 69 benchmarks, 36 instances generate more than 10^6 clauses under direct De Morgan expansion, and the maximum clause width reaches 500. In contrast, our probabilistic Tseytin protocol, combined with the balanced translation (Section 3.5.1) using $\omega = 100$, produces at most 4,342 clauses for the translated invariant component and bounds every introduced clause width to at most $\omega + 1 = 101$. Since ZKP performance depends on both clause count and maximum width, this demonstrates why our implication-checking protocol is necessary for practical ZK performance.

4.7.3 Cheat probability of ZKUNSAT-IN. We also evaluate the cheat probability in ZKUNSAT-IN in our benchmarks, where 9 designs required no fresh auxiliary variables ($M = 0$) to encode the implication since their invariant is a unit clause. The largest p_{cheat} was observed on `ILA_Piccolo_BEQ_sanity`, at 4.3505×10^{-10} for $N = 2^{60}$, which remains small.

5 Comparison

In the hardware community, the most closely related works are Pythia [42] and MP$\ell\circ$C [43]. Both methods are restricted to verifying the correctness of single execution traces on the combinational ISCAS'85 benchmarks. MP$\ell\circ$C [43] introduces additional structural overhead to logic-lock the design. Although the computation completes within seconds, the locked netlist must be disclosed to the verifier, exposing the design to potential structural inference and logic-locking attacks [7]. Pythia [42] compiles circuits into a ZK-friendly format, and its experimental results show that, with 748 GB of memory, the evaluated design `c7552`, which contains a few thousand gates, requires about 500 s. In contrast, within the same time budget, our approach can verify the design `a10-p53`, which contains 308,000 gates.

In the formal methods domain, CTL model checking [36] could model hardware designs as Kripke structures and incurs a verification cost of $O(mn^2)$, where n denotes the number of states

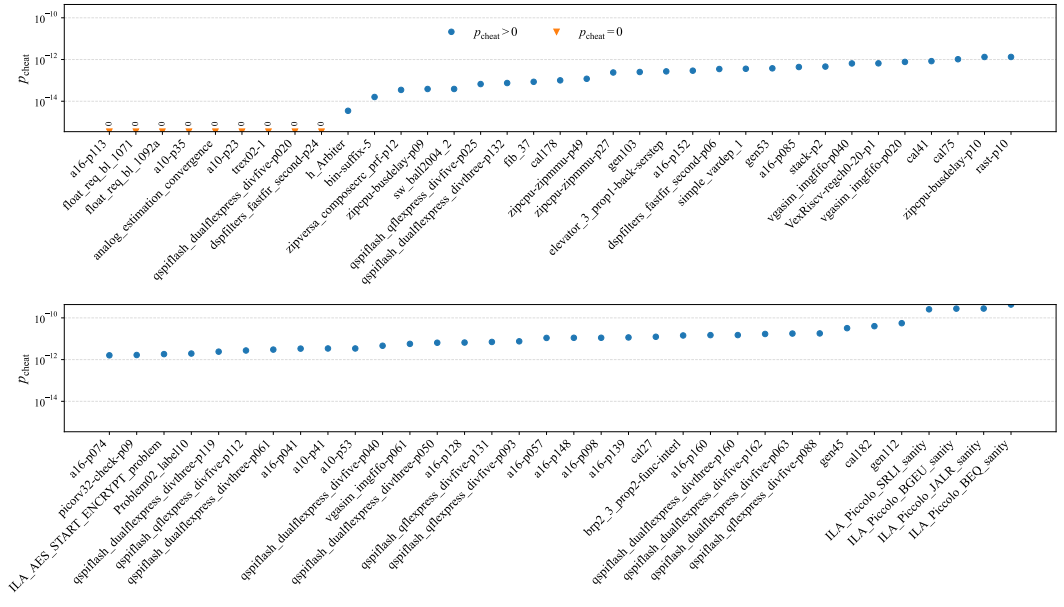


Fig. 10. Cheat probability p_{cheat} per design with the maximal invariant width set to 100 and $N = 2^{60}$. Nine designs require no fresh auxiliary variables ($M = 0$) to encode the implication. The largest probability occurs for `ILA_Piccolo_BEQ_sanity`, at 4.3505×10^{-10}, which is still relatively small.

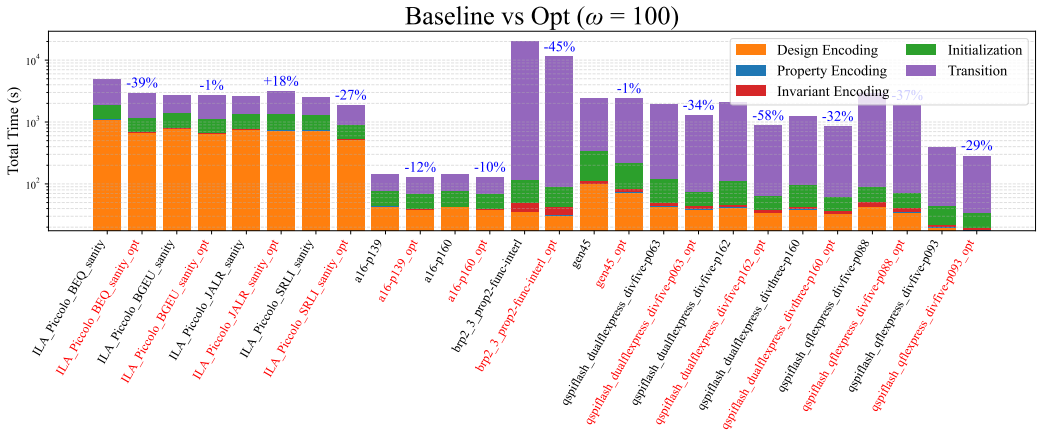


Fig. 11. Comparison of baseline vs. optimized (set maximal clause width of invariant to 100) runtime across 69 benchmark instances. The stacked bars break down the runtime into design encoding, property encoding, invariant encoding, initialization, and transition. Percentage labels above bars indicate performance improvement compared to the baseline(non-optimized). Among the 69 cases, 12 showed performance improvement, 1 showed degradation, and 56 exhibited no change.

and m denotes the number of CTL operators. Even with a single CTL operator, a design with 64 states requires around 750 s. It is therefore infeasible to verify a design such as `a10-p53`, which has 41,262 latches (corresponding to a state space of $2^{41,262}$).

We report the total time required to prove safety in the plaintext setting using a comparable Python implementation of the same algorithm in Figure 12. The total proof time excludes the

unnecessary step of design encoding and subsequent transformations. The cost of proving safety scales linearly with the number of proof steps, with a maximum proving time of 27.6s and an average of 1.3s. We consider this overhead acceptable in light of the privacy guarantees provided and in comparison with prior approaches, since verification is performed once per design and the resulting certificate is bound to the committed design.

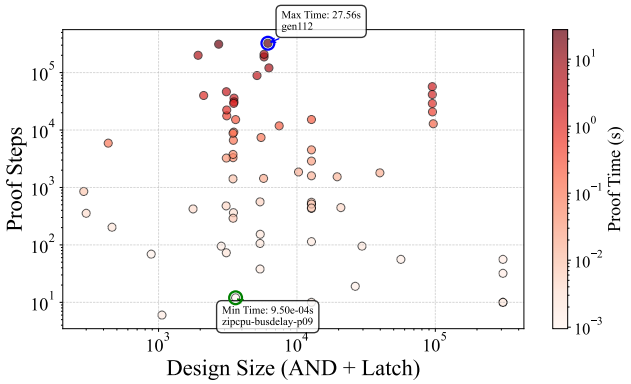


Fig. 12. In the plaintext setting, the designer only needs to prove the correctness of the initialization and transition, with a maximum runtime of 27.6s and an average runtime of 1.3s.

6 Conclusion

We propose ZSafe, the first ZKP–based framework that enables a designer to prove the safety of a hardware design by induction without disclosing any design internals to an untrusted verifier. In the hardware domain, our framework addresses confidentiality leakage during design distribution in globalized hardware supply chains and demonstrates scalability through evaluation on the HWMCC'24 benchmarks, which comprise non-trivial safety properties over complex, large-scale designs. In the formal methods domain, ZSafe advances the theory and practice of inductive safety verification under restricted observability. It provides the first end-to-end framework that compiles inductive safety proofs over Boolean transition systems into ZKP, while preserving soundness of induction. We believe ZSafe applies to any security-critical verification task reducible to inductive safety checking over Boolean transition systems, such as software verification, protocol verification, and distributed system correctness, particularly in cloud-based or outsourced verification settings where the verifier is untrusted and may execute or simulate the system prior to formally verifying it. Although ZSafe incurs higher cryptographic overhead than plaintext verification, we consider this cost a one-time expense justified by the need for strong privacy guarantees beyond what existing approaches support.

6.1 Scope and Limitations

The current implementation of ZSafe focuses on safety properties over Boolean transition systems represented as sequential AIGs. This design choice matches the standard setting of SAT-based hardware safety verification and allows ZSafe to leverage existing AIG-based hardware verification infrastructure. ZSafe assumes that the designer can obtain a suitable inductive invariant using external tooling, and invariant generation is orthogonal to our ZK proof framework.

Properties or models outside this Boolean safety setting can still be handled when they admit a reduction to the supported form. For example, certain liveness properties can be reduced to safety

using standard liveness-to-safety transformations [11]. Similarly, word-level verification tasks, such as datapath verification or SMT-level abstractions, may currently be supported through bit-blasting or external preprocessing. Extending ZSafe with more direct support for higher-level theories [38] is an important direction for future work.

6.2 Related Work

IP Privacy Hardware IP protection is a fundamental concern in modern global supply chains [35, 46], where designs must be shared with multiple, potentially untrusted parties, and any breach can lead to severe financial losses [47–49]. To address confidentiality leakage during design distribution, traditional methods such as obfuscation [21] and watermarking [17] have been widely studied to deter post-distribution misuse. Watermarking allows the designer to embed a special identifier for ownership attribution. Obfuscation helps the designer conceal design internals to restrict correct usage to authorized users. However, these methods do not address privacy concerns for untrusted third-party verifiers, who may misuse the design under the guise of integration or verification.

IEEE-1735 [2] mitigates this issue by explicitly assuming a trusted EDA toolchain, a model widely adopted in industry [5, 32]. It enables the designer to encrypt proprietary designs and permits the untrusted verifier to perform simulation or synthesis without revealing the underlying design. However, the trusted third-party assumption is overly strong, as evidenced by previous *Sherlocking* accusations [44], and IEEE-1735 itself is vulnerable under realistic attacks [16, 53]. Later work eliminates the need for trusted third parties by leveraging advanced cryptographic primitives. MP$\ell\circ$C [43] enables an untrusted verifier to simulate an obfuscated design, where deobfuscation is performed via secure multiparty computation using a secret key held by the designer. Pythia [42] allows the designer to prove the correctness of the simulation in zero knowledge for public test inputs provided by the verifier. Both methods were evaluated on ISCAS'85 benchmarks containing only a few thousand gates. Despite these advances, the aforementioned approaches do not provide formal guarantees.

Privacy-Preserving Formal Verification Efforts to address privacy concerns posed by untrusted third parties in formal verification span SAT solving [40], proving UNSAT [38, 39], CTL model checking [36], and runtime verification [8, 30, 56]. Our work adopts a static model-checking approach to safety verification and complements runtime verification. We focus on verification over Boolean transition systems, targeting properties expressible in Linear Temporal Logic (LTL) [45].

However, privacy-preserving safety verification remains relatively underexplored in this domain. CTL model checking [36] requires modeling the system as Kripke structures, leading to explicit state-space construction and state explosion. SAT-based model checking partially mitigates this issue by reasoning over Boolean transition systems [14, 31]. However, existing SAT-based methods lack support for verification over transition systems. For example, ppSAT [40] enables two-party privacy-preserving SAT solving over private Boolean formulas but incurs significant cryptographic overhead. Certifying the validity of logical formulas in zero knowledge has been shown to be scalable [38, 39], demonstrating the feasibility of applying ZKP techniques to non-trivial formal reasoning tasks. Neither is directly applicable to stateful reasoning. Safety verification requires stateful reasoning over a complete transition system, where correctness is established through inductive arguments over reachable states. This gap motivates ZSafe, a zero-knowledge framework for safety verification over Boolean transition systems. Our evaluation in Section 4.3 further demonstrates that ZSafe scales to hardware designs that are orders of magnitude larger than prior privacy-preserving hardware verification methods [36, 42, 43].

7 Data-Availability Statement

7.1 Artifact

Our artifact includes the full implementation of our ZSafe for evaluation, with optimization of Balanced Tseytin Translation ($\omega = 100$) (Section 3.5.1) and Strengthened Invariant (Section 3.5.2) as well as all scripts and benchmarks used in our evaluation. We include scripts to collect all necessary data to recreate the experiment figure and a small demo to demonstrate a single verification procedure.

We use a modified version of **rIC3** in our experiments. To simplify the evaluation process for reviewers, we precompute and include all necessary inductive invariants in our dataset, eliminating the need to install and invoke the model checker during evaluation.

7.2 Evaluation

The artifact is available at https://anonymous.4open.science/r/ZsafeArtifact-254C/README.md. All required software dependencies are listed in the provided setup instructions. Please refer to the accompanying `README.md` for detailed installation steps and configuration guidelines. The estimated total time cost is approximately 37 hours, of which the four most complex benchmarks account for about 27 hours.

References

[1] [n. d.]. MCHyper: A hardware model checker for hyperproperties. https://github.com/reactive-systems/MCHyper. Accessed: 2026-01.

[2] 2015. IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP). *IEEE Std 1735-2014 (Incorporates IEEE Std 1735-2014/Cor 1-2015)* (2015), 1–90. https://doi.org/10.1109/IEEESTD.2015.7274481

[3] 2024. Hardware Model Checking Competition. https://hwmcc.github.io/.

[4] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 191–219.

[5] AMD Xilinx. 2024. *Understanding IEEE 1735 Structural Elements*. AMD. https://docs.amd.com/r/en-US/ug1118-vivado-creating-packaging-custom-ip/Understanding-IEEE-1735-Structural-Elements

[6] Semiconductor Industry Association et al. 2022. Global Semiconductor Sales, Units Shipped Reach All-Time Highs in 2021 as Industry Ramps Up Production Amid Shortage. *Haettu* 16 (2022), 2022.

[7] Kimia Zamiri Azar, Hadi Mardani Kamali, Houman Homayoun, and Avesta Sasan. 2019. SMT attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the SAT attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), 97–122.

[8] Ryotaro Banno, Kotaro Matsuoka, Naoki Matsumoto, Song Bian, Masaki Waga, and Kohei Suenaga. 2022. Oblivious online monitoring for safety LTL specification via fully homomorphic encryption. In *International Conference on Computer Aided Verification*. Springer, 447–468.

[9] A Biere. [n. d.]. BooleForce and TraceCheck (2014).

[10] Armin Biere. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modelling and Computation* 4, 2-4 (2008), 75–97.

[11] Armin Biere, Cyrille Artho, and Viktor Schuppan. 2002. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science* 66, 2 (2002), 160–177.

[12] Armin Biere, Keijo Heljanko, and Siert Wieringa. 2011. AIGER 1.9 and beyond. (2011).

[13] Xavier Bonnetain. 2019. Collisions on feistel-mimc and univariate gmimc. (2019).

[14] Aaron R Bradley. 2012. Understanding ic3. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 1–14.

[15] Katharina Ceesay-Seitz, Flavien Solt, and Kaveh Razavi. 2024. μCFI: Formal Verification of Microarchitectural Control-flow Integrity. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 213–227.

[16] Animesh Chhotaray, Adib Nahiyan, Thomas Shrimpton, Domenic Forte, and Mark Tehranipoor. 2017. Standardizing Bad Cryptographic Practice: A Teardown of the IEEE Standard for Protecting Electronic-Design Intellectual Property. In *in ACM SIGSAC CCS '17*. 1533–1546.

[17] Aijiao Cui, Chip-Hong Chang, Sofiene Tahar, and Amr T Abdel-Hamid. 2011. A robust FSM watermarking scheme for IP protection of sequential circuit design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 5 (2011), 678–690.

[18] Rohit Dureja, Arie Gurfinkel, Alexander Ivrii, and Yakir Vizel. 2021. IC3 with internal signals. In *2021 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 63–71.

[19] Nicole Fern, Ismail San, and Kwang-Ting Tim Cheng. 2017. Detecting hardware trojans in unspecified functionality through solving satisfiability problems. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 598–504.

[20] Bernd Finkbeiner, Markus N Rabe, and César Sánchez. 2015. Algorithms for model checking HyperLTL and HyperCTL. In *International Conference on Computer Aided Verification*. Springer, 30–48.

[21] Domenic Forte, Swarup Bhunia, and Mark M Tehranipoor. 2017. *Hardware protection through obfuscation*. Springer.

[22] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1991. Proofs That Yield Nothing But Their Validity Or All Languages in NP Have Zero-Knowledge Proof Systems. *J. ACM* 38, 3 (July 1991), 691–729. https://doi.org/10.1145/116825.116852

[23] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. 2008. Delegating computation: interactive proofs for muggles. 113–122. https://doi.org/10.1145/1374376.1374396

[24] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). 291–304. https://doi.org/10.1145/22145.22178

[25] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*. 519–535.

[26] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. 2023. Poseidon2: A faster version of the poseidon hash function. In *International Conference on Cryptology in Africa*. Springer, 177–203.

[27] Jens Groth. 2010. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. 321–340. https://doi.org/10.1007/978-3-642-17373-8_19

[28] Xiaolong Guo, Raj Gautam Dutta, Jiaji He, and Yier Jin. 2017. Pch framework for ip runtime security verification. In *2017 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 79–84.

[29] Mohammad Hashemi, Steffi Roy, Fatemeh Ganji, and Domenic Forte. 2022. Garbled EDA: Privacy Preserving Electronic Design Automation. In *2022 IEEE/ACM International Conference On Computer Aided Design (ICCAD '22)*. 1–9. To appear..

[30] Thomas A Henzinger, Mahyar Karimi, and KS Thejaswini. 2025. Privacy-Preserving Runtime Verification. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 2774–2787.

[31] HWMCC Organization. 2024. Hardware Model Checking Competition 2024. https://hwmcc.github.io/2024/.

[32] Intel Corporation. 2024. *Support for the IEEE 1735 Encryption Standard*. Intel. https://docs.altera.com/r/docs/683102/current

[33] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2007. Zero-knowledge from secure multiparty computation. 21–30. https://doi.org/10.1145/1250790.1250794

[34] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. 955–966. https://doi.org/10.1145/2508859.2516662

[35] Yier Jin. 2015. Introduction to hardware security. *Electronics* 4, 4 (2015), 763–784.

[36] Samuel Judson, Ning Luo, Timos Antonopoulos, and Ruzica Piskac. 2020. Privacy Preserving CTL Model Checking through Oblivious Graph Algorithms. In *Proceedings of the 19th Workshop on Privacy in the Electronic Society (WPES@CCS) '19*.

[37] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E Culler. 2019. {JEDI}:{Many-to-Many}{End-to-End} encryption and key delegation for {IoT}. In *28th USENIX security symposium (USENIX Security 19)*. 1519–1536.

[38] Daniel Luick, John C Kolesar, Timos Antonopoulos, William R Harris, James Parker, Ruzica Piskac, Eran Tromer, Xiao Wang, and Ning Luo. 2024. {ZKSMT}: A {VM} for Proving {SMT} Theorems in Zero Knowledge. In *33rd USENIX Security Symposium (USENIX Security 24)*. 3837–3845.

[39] Ning Luo, Timos Antonopoulos, William R Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. 2022. Proving UNSAT in zero knowledge. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 2203–2217.

[40] Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. 2022. ppSAT: Towards Two-Party Private SAT Solving. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2983–3000. https://www.usenix.org/conference/usenixsecurity22/presentation/luo

[41] Xingyu Meng, Shamik Kundu, Arun K Kanuparthi, and Kanad Basu. 2021. Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2021), 466–477.

[42] Dimitris Mouris and Nektarios Georgios Tsoutsos. 2020. Pythia: Intellectual Property Verification in Zero-Knowledge. In *2020 57th ACM/IEEE Design Automation Conference (DAC '20)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218639

[43] Dimitris Mouris, Charles Gouert, and Nektarios Georgios Tsoutsos. 2023. MPloC: Privacy-Preserving IP Verification using Logic Locking and Secure Multiparty Computation. *Cryptology ePrint Archive* (2023).

[44] NPR. 2024. Apple just made your app obsolete? You've been 'Sherlocked'. *NPR* (17 June 2024). https://www.npr.org/2024/06/17/g-s1-4912/apple-app-store-obsolete-sherlocked-tapeacall-watson-copy

[45] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th IEEE Annual Symposium on Foundations of Computer Science (FOCS '77)*.

[46] Jeyavijayan JV Rajendran. 2017. An overview of hardware intellectual property protection. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 1–4.

[47] Reuters. 2010. Motorola sues Huawei, Lemko for trade secret theft. *Reuters* (22 July 2010). https://www.reuters.com/article/technology/motorola-sues-huawei-lemko-for-trade-secret-theft-idUSN21191382/

[48] Reuters. 2014. U.S. court voids $920 million DuPont award in Kevlar case. *Reuters* (3 April 2014). https://www.reuters.com/article/business/us-court-voids-920-million-dupont-award-in-kevlar-case-idUSBREA321FE/

[49] Reuters. 2016. Epic Systems wins $940 million U.S. jury verdict in Tata trade secret case. *Reuters* (17 April 2016). https://www.reuters.com/article/world/epic-systems-wins-940-million-us-jury-verdict-in-tata-trade-secret-case-idUSKCN0XE02A/

[50] Reuters Staff. 2024. Former Russian ASML employee set for court hearing in alleged IP theft case. *Reuters* (2024). https://www.reuters.com/technology/former-russian-asml-employee-set-court-hearing-alleged-ip-theft-case-2024-12-09/ Accessed: 2025-04-02.

[51] Julian D Schwab, Silke D Kühlwein, Nensi Ikonomi, Michael Kühl, and Hans A Kestler. 2020. Concepts in Boolean network modeling: What do they all mean? *Computational and structural biotechnology journal* 18 (2020), 571–582.

[52] Semiconductor Industry Association, Anti-Counterfeiting Task Force. 2018. Combatting Counterfeit Semiconductors. https://www.semiconductors.org/wp-content/uploads/2018/06/ACTF-Whitepaper-Counterfeit-One-Pager-Final.pdf. White Paper.

[53] Julian Speith, Florian Schweins, Maik Ender, Marc Fyrbiak, Alexander May, and Christof Paar. 2022. How Not to Protect Your IP–An Industry-Wide Break of IEEE 1735 Implementations. In *2022 IEEE Symposium on Security and Privacy (Oakland '22)*. IEEE, 1656–1671.

[54] Yuheng Su, Qiusong Yang, Yiwei Ci, Tianjun Bu, and Ziyu Huang. 2025. The rIC3 Hardware Model Checker. *arXiv preprint arXiv:2502.13605* (2025).

[55] Bao Tran. [n. d.]. Chip Manufacturing Costs in 2025–2030: How Much Does It Cost to Make a 3nm Chip? https://patentpc.com/blog/chip-manufacturing-costs-in-2025-2030-how-much-does-it-cost-to-make-a-3nm-chip.

[56] Masaki Waga, Kotaro Matsuoka, Takashi Suwa, Naoki Matsumoto, Ryotaro Banno, Song Bian, and Kohei Suenaga. 2024. Oblivious Monitoring for Discrete-Time STL via Fully Homomorphic Encryption. In *International Conference on Runtime Verification*. Springer, 59–69.

[57] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty computation toolkit.

[58] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, Vol. 97.

[59] Qizhi Zhang, Liang Liu, Yidong Yuan, Zhe Zhang, Jiaji He, Ya Gao, Yao Li, Xiaolong Guo, and Yiqiang Zhao. 2022. A Gate-Level Information Leakage Detection Framework of Sequential Circuit Using Z3. *Electronics* 11, 24 (2022), 4216.

[60] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. 2018. End-to-end automated exploit generation for validating the security of processor designs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 815–827.