ZEKRA: Zero-Knowledge Control-Flow Attestation

Heini Bergsson Debes
Technical University of Denmark (DTU)
heib@dtu.dk

Edlira Dushku
Aalborg University
edu@es.aau.dk

Thanassis Giannetsos
Ubitech Ltd.
agiannetsos@ubitech.eu

Ali Marandi
Technical University of Denmark (DTU)
alimar@dtu.dk

ABSTRACT

To detect runtime attacks against programs running on a remote computing platform, Control-Flow Attestation (CFA) lets a (trusted) verifier determine the legality of the program's execution path, as recorded and reported by the remote platform (prover). However, besides complicating scalability due to verifier complexity, this assumption regarding the verifier's trustworthiness renders existing CFA schemes prone to privacy breaches and implementation disclosure attacks under "honest-but-curious" adversaries. Thus, to suppress sensitive details from the verifier, we propose to have the prover outsource the verification of the attested execution path to an intermediate worker of which the verifier only learns the result. However, since a worker might be dishonest about the outcome of the verification, we propose a purely cryptographical solution of transforming the verification of the attested execution path into a *verifiable computational task* that can be reliably outsourced to a worker without relying on any trusted execution environment. Specifically, we propose to express a program-agnostic execution path verification task inside an arithmetic circuit whose correct execution can be verified by untrusted verifiers in zero knowledge.

CCS CONCEPTS

• **Security and privacy → Security protocols**; **Privacy-preserving protocols**; *Software security engineering*.

KEYWORDS

Control-Flow Attestation; Verifiable Computation; zkSNARK

ACM Reference Format:
Heini Bergsson Debes, Edlira Dushku, Thanassis Giannetsos, and Ali Marandi. 2023. ZEKRA: Zero-Knowledge Control-Flow Attestation. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3579856.3582833

1 INTRODUCTION

To safeguard the increasing computing system attack landscape, traditional remote attestation schemes let a (trusted) verifier reason about the state of a remote prover's computing platform. The security of such schemes generally relies on a trust anchor on the prover, capable of securely recording and authenticating platform evidence. Building on this concept, Control-Flow Attestation (CFA) aims to determine whether a program was executed correctly on a resource-constrained prover by verifying that no runtime attacks (e.g., ROP [47]) subverted the program's control-flow behavior. To ensure that a program was executed correctly, existing CFA schemes assume a trusted verifier who maintains complete reference materials, such as the program's Control-Flow Graph (CFG) and in-memory program layout, and other acceptance criteria to decide on the legality of the attested program's execution path, as recorded and reported by the prover's trust anchor. However, besides impairing scalability due to the verifier complexity, the unattractive need to exchange comprehensive prover information discourages the adoption of CFA in public-facing and emerging multi-domain services [4].

While not yet demonstrated for CFA, the concept of Property-Based Attestation (PBA) [13] could reduce the verifier complexity and prevent information disclosure by giving the verifier only the verification result in the form of some semantical property. However, performing the necessary verification and property translation locally on the prover would require a resourceful trust anchor capable of correctly maintaining all trusted reference materials *and performing the verification correctly*, which is sometimes impractical, especially for resource-constrained settings such as those generally considered in CFA. Here, the provers are severely underpowered devices equipped with carefully designed minimalistic trust anchors [15] whose sole purpose is to record and authenticate a program's execution path during attestation. Therefore, without complicating the prover, another option is to introduce an intermediate, more powerful party, which we refer to as a worker (sometimes called an "attestation proxy"), responsible for performing the attestation verification on behalf of the verifier and conveying only the result back to the verifier. However, to convince the verifier that the verification was done correctly, we would again generally encounter heavy assumptions, such as requiring trusted hardware or a Trusted Execution Environment (TEE) with attestable execution (e.g., Intel SGX) to protect the verification process on the worker.

Instead, we propose to utilize Verifiable Computation (VC) to transform the task of verifying the prover's attested program execution into an outsourceable arithmetic circuit whose *proof of correct execution* can be generated by the worker and efficiently verified by the verifier, proving the attestation verification's correctness and outcome. Using VC, we *need no assumption of any trusted computing base on the worker* while also protecting against a wider range of attackers than approaches that consider some form of TEE. Further, to hide certain inputs of the proof generation (e.g., the attested execution path and program details) from *completely untrusted* verifiers, we use a privacy-enhanced VC scheme called zero-knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARK).

Contributions. We propose a novel protocol called *ZEro-Knowledge contRol-flow Attestation* (ZEKRA), which is, to our knowledge, the first privacy-preserving CFA protocol. Without imposing additional prover assumptions, we remove all trust and complexity assumptions regarding verifiers by outsourcing attestation verifications to intermediate workers who employ VC to convince verifiers about the verification results. Our work offers the following contributions: (i) We present a novel scheme that lets underpowered provers convince untrusted verifiers about a program's correct execution in zero-knowledge by offloading the verification to an intermediate worker that assures verifiers about the result without disclosing any secrets using zkSNARK technology; (ii) We detail our outsourceable circuit design, including the use of several circuit optimization techniques; (iii) Realistic case studies, showing how ZEKRA can resolve privacy issues in privacy-sensitive (*non*-time-critical) application domains; and, (iv) We validate and benchmark ZEKRA with a proof-of-concept implementation, which we make publicly available [14] to ensure reproducibility and encourage further work.

2 RELATED WORKS

Many prevention methods exist against program runtime attacks, e.g., shadow stacks and stack canaries, Control-Pointer Integrity (CPI), and Control-Flow Integrity (CFI) [1]. However, these methods fail to provide any assurance to a remote entity since all enforcement happens locally and will (depending on the enforced policy) abruptly stop–and possibly crash–a device upon violations, which can be dangerous in certain safety-critical applications.

Runtime attestation. To *detect* program control-flow attacks, C-FLAT [2] proposed instrumenting the program to self-report all control-flow events to a TEE during runtime, where, once hashed, trusted verifiers can check if the digest exists in a set of trusted reference values. LO-FAT [15] leverages a customized hardware module to intercept the executed instructions at runtime to improve C-FLAT's performance. ATRIUM [53] enhances both C-FLAT and LO-FAT to detect TOCTOU attacks that swap malign program segments with benign segments during attestation to evade detection. ATRIUM relies on a customized hardware module that runs attestation parallel to the main processor. ScaRR [50] aims to apply CFA to complex systems, e.g., cloud-native virtual machines. To deal with the challenge of representing all the valid execution paths in complex systems, ScaRR follows C-FLAT's approach of splitting the control-flow execution into sub-paths, where the idea is basically to record each unique loop path only once in the execution path while maintaining associated counters to track the number of times each path was taken. Tiny-CFA [41] is a CFA protocol that relies on the APEX Proof-of-Execution (PoX) architecture [40] and, similar to C-FLAT, assumes that the software is instrumented. ReCFA [54] performs control-flow attestation of complex software by relying on the static binary analysis and binary instrumentation. In particular, the scheme compresses the control-flow evidence efficiently and enforces control-flow integrity policy at the binary level with a remote shadow stack. Note that other CFA schemes also exist, which additionally consider: utilizing machine learning [28], a log-based approach and use of physically unclonable functions [37], distributed settings and use of multiset hash function to reduce the size of the reported execution path [3]. Some approaches also consider detecting data-oriented attacks [16, 34, 49] by verifying the integrity of both control-flow and data involved in the execution.

Our work is *complimentary to the above approaches*: prior work can leverage our scheme to weaken the verifier trust assumptions. Specifically, whereas prior CFA works generally focus on recording and reporting the program's execution path, we focus on the layer between provers and verifiers to remove the *omniscient and trusted verifier* assumption by making the verification zero-knowledge.

Verifiable computation. Unlike CFA schemes that consider underpowered provers and powerful verifiers, proof-based VC enables weak verifiers (our provers) to outsource computationally intensive computations to powerful yet untrusted provers (our workers) who return proof that the computation was done correctly. Moreover, to enable secret inputs in the computations, privacy-enhanced VC schemes, e.g., zkSNARKs [23], guarantee that the proof reveals nothing about the secret inputs. Furthermore, whereas proof generation is slow, verification is remarkably fast, making zkSNARKs attractive, especially in Distributed Ledger Technology (DLT) [43].

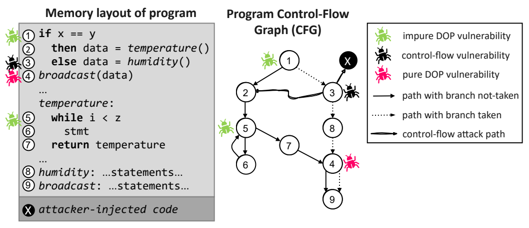
To verify general programs using zkSNARKs, specialized compilers, such as TinyRAM [5], vnTinyRAM [6], and Buffet [51] have enabled the transformation of traditional programs into low-level circuits, whose execution can be proven and verified securely. For example, the TinyRAM [5] circuit compiler takes a high-level C program and a time-bound T as input and compiles the program to special assembly instructions, whose emulated execution on some input for up to T cycles in a general-purpose MIPS-like CPU, called TinyRAM, is expressed as an arithmetic circuit that verifies the correct execution of the input program. However, the principal disadvantage is cost since the number of circuit constraints (i.e., the size of the circuit) grows unwieldy as the program complexity increases, which strongly correlates to the amount of time it takes to generate proof over the circuit's execution. Improving on the per-cycle cost, TinyRAM's successor, vnTinyRAM [6], achieved a quasi-constant per-cycle cost of $\approx 1,458$ constraints, and later Buffet [51] further improved control flow and random memory access by 1-3 and 1-2 orders of magnitude, respectively.

Nonetheless, directly expressing general programs as circuits remains expensive. Fortunately, CFA schemes have demonstrated that verifying a program's execution is enough to convince a remote verifier that a program executed correctly with respect to its control flow. While CFA's security guarantee is only a subset of that of VC, this paper demonstrates that combining the two allows underpowered provers to prove a program's execution correctness.

3 BACKGROUND

3.1 Program Composition

A compiled program's code can be represented by its Control-Flow Graph (CFG), which encapsulates all possible program executions by modeling the legal control flow between all of the program's statements. However, since not all statements affect the control flow, we typically fractionate the statements into maximal-length sequences of branchless statements that ultimately end in a branch,



```
1 if x == y
2   then data = temperature()
3   else data = humidity()
4 broadcast(data)
  ...
  temperature:
5   while i < z
6     stmt
7   return temperature
  ...
8 humidity: ...statements...
9 broadcast: ...statements...
X attacker-injected code
```

Figure 1: Abstract view of a program's CFG and threats.

jump, or predicated operation. We denote each such sequence as a basic block (BBL) and have the CFG model the control flow only between program BBLs. Let $G = (N, E)$ denote a directed graph (CFG), where nodes $n_i \in N$ correspond to BBLs and edges $e = (n_i, n_j) \in E$ denote possible transfers of control. We refer to edges corresponding to (direct and indirect) jumps and calls as forward edges and returns as back edges. We further label any node n_i an entry node ($n_{\triangleright}$) if it is unreachable (i.e., has indegree $\delta^-(n_i)$ of zero) and a final node ($n_{\blacktriangleleft}$) if it has no reachable nodes (i.e., has outdegree $\delta^+(n_i)$ of zero). (We denote with $\Delta(G)$ the maximum *outdegree* of G.) Finally, any continuous edge sequence is a legal execution path if it connects some $n_{\triangleright}$ to $n_{\blacktriangleleft}$ (denoted $n_{\triangleright} \rightsquigarrow n_{\blacktriangleleft}$).

3.2 Runtime Attacks

We continue with a description of major runtime attack classes that can induce harmful behavior by exploiting software vulnerabilities to corrupt a program's control and data planes. As a running example, let us consider the simple program skeleton in Fig. 1.

Control-based attacks. The most common attacks target a program's control plane to execute unintended code by explicitly diverting its execution path. There are essentially two variants: code injection and code reuse. With code injection, an adversary crafts and injects a payload into memory and redirects a benign program's control flow to execute the payload. As an example, consider that in Fig. 1 an adversary has injected node n_X and diverted the control-flow from (n_3, n_8) to (n_3, n_X), resulting in the execution of code in n_X instead of n_8. However, being an early attacking methodology, code injection is easily defeated using common mechanisms such as Data-Execution Prevention (DEP). For the latter variant, however, it gets more difficult. Without injecting code, code-reuse attacks reuse existing program code to achieve some unintended behavior–using control plane maneuvers such as Return-Oriented Programming (ROP) [47] and Jump-Oriented Programming (JOP) [9].

With ROP/JOP, an adversary fabricates a new program by stitching together a chain of benign pieces of existing code (gadgets) that end in function returns (ROP) or indirect jumps or function calls (JOP). The chain is then written into memory (e.g., through a stack overflow vulnerability), where, once it is triggered (e.g., by replacing a function's return address with that of the first gadget), the gadgets execute in sequence. For example, in Fig. 1, the adversary launches a ROP attack diverting the control-flow from (n_3, n_8) to (n_3, n_2) to execute code in the other branch.

Non-control-data attacks. Another class of attacks exists, which corrupt data variables to make programs yield unexpected outputs or indirectly drive program execution down unexpected or unauthorized paths. The attacking methodology behind non-control-data attacks is the application of Data-Oriented Programming (DOP) [27] which we can call impure or pure, depending on whether the program execution path is influenced (impure) or only data variables are altered with no effect on the path (pure). However, due to the difficulty of effectively verifying a program's data flow at the verifier, CFA schemes generally disregard such attacks.

3.3 Toward CFA in Zero-Knowledge

Zero-knowledge proofs [21, 30, 38] enable a prover to convince a verifier that a statement is true by demonstrating knowledge of a satisfying witness without revealing anything about the witness.

Path explosion problem. Two fundamental zero-knowledge proof system constructions are range proofs [31] and set membership proofs [8], respectively. With range proofs, we can prove knowledge of a secret s by demonstrating that s belongs to the interval $[u, v)$ for arbitrary integers u and v. With set membership, we can prove that s belongs to an arbitrary set S. In both cases, however, we need a clearly defined interval or set during the proof system instantiation. Therefore, for us to use either approach on the worker to prove the validity of an attested execution path, we would need to know all legal execution paths, which is generally impractical.

For a directed *acyclic* graph (DAG), we could enumerate all paths by performing a depth-first search. However, for a CFG, a directed graph, the set of paths is unbounded, which we can illustrate by considering the program in Fig. 1. For example, if the program loop is only used as a busy-wait, the set of paths grows unwieldy since any number of iterations constitutes a unique path. Moreover, as the complexity of the program increases, so does the number of paths. Therefore, we must sacrifice precision if we insist on a finite set of paths. One approach is simply transforming the CFG into a DAG by pruning all back edges. Similarly, as described in [26], we can identify all strongly connected components, i.e., maximal sets of nodes where a path exists between any two nodes in a set, contract each component into a single node (e.g., fuse n_5 and n_6 in Fig. 1) to form a condensation graph (which is acyclic), and then consider only the paths within this graph. However, both approaches are imperfect as the coarse-grained granularity of condensation or ignorance when discarding back edges leaves attacks undetectable.

zkSNARKs. To let provers attest to arbitrary execution paths without enumerating paths beforehand or sacrificing precision, we opt to use zero-knowledge Succinct Non-interactive Arguments of Knowledge (zkSNARK) [23, 43], where we create a program that accepts a program's CFG together with any path as input and verifies that the path is legal according to the CFG. We then transform the program into a low-level arithmetic circuit C representation [6] over a finite field $\mathbb{F}$ (typically a 254-bit prime field $\mathbb{F}_p$) composed of additions and multiplications $\bmod p$. Given such a circuit, we can instantiate a zkSNARK proof system that lets provers attest to arbitrary execution paths, which can be proven correct by generating a proof π that the circuit was satisfied when executed on the attested path.

Specifically, let C be our arithmetic circuit. A zkSNARK allows the worker to prove that she correctly executed C on public input x and secret input $\overline{u}$ (bar denotes *secret* input), as follows. After taking C as input, a trusted party conducts a one-time setup that gives two public keys: a proving key pk and a verification key vk. The proving key pk enables any untrusted worker to produce a proof π attesting to the fact that x and $\overline{u}$ satisfied C. The non-interactive proof π is *zero knowledge* and a *proof of knowledge*. The proof reveals nothing about $\overline{u}$, but anyone can verify its correctness using only vk.

In total, zkSNARK schemes consist of three algorithms:

- $(\mathsf{pk}_C, \mathsf{vk}_C) \leftarrow \mathsf{KeyGen}(C, 1^\lambda)$: given a circuit C, output pk_C and vk_C as the public proving and verification keys.
- $(y, \pi) \leftarrow \mathsf{Prove}(C, \mathsf{pk}_C, x, \overline{u})$: given a circuit C, proving key pk_C, public x and secret $\overline{u}$ inputs, output $y \leftarrow C(x, \overline{u})$, and the proof π of the computation correctness.
- $\{0, 1\} \leftarrow \mathsf{Verify}(\mathsf{vk}_C, x, y, \pi)$: given a verification key vk_C and statement (x, y), output 1 only if $y = C(x, \overline{u})$.

In most constructions, C is expressed in the NP-complete languages called Rank-1-Constraint-Systems (R1CS) and Quadratic Arithmetic Programs (QAPs) [20]. In R1CS, computations are encoded as a set of conditions over its variables such that correct execution equals finding a satisfying variable assignment, whereas, with QAP, computations are instead represented as a set of quadratic equations. However, as with any VC protocol that requires the computation task to be expressed as arithmetic circuits over some field $\mathbb{F}_p$, the size of the sets (i.e., constraints) corresponds to the circuit size and determines the time needed to generate proofs.

4 SYSTEM AND THREAT MODEL

System model. We consider a network of four main entities:

1. **Prover** is an untrusted and underpowered device equipped-with a minimal trust anchor capable of tracing and authenticating a program's execution. Note that these two capabilities constitute the minimum trusted computing base for guaranteeing the security of the attestation and thus restrict the trust anchor's influence and performance footprint on the prover as much as possible. See Appendix A for further clarification on the underlying trust assumptions. However, note that the choice of tracing via (i) interfacing with the CPU's pipeline [15, 16, 53], or (ii) having instrumented programs, stored in DEP-enabled memory, self-report control-flow transfers [2, 3, 49, 50], is considered complementary.
2. **Verifier** is an untrusted device wishing to check the correctness of a program (or part thereof) executed on the prover.
3. **Worker** is a semi-untrusted and computationally capable device that generates zkSNARK proofs for convincing untrusted verifiers about the correctness of a prover's attested execution paths without disclosing any secret inputs. See Appendix A for further clarification on the worker's role.
4. **Network operator** is trusted to execute the KeyGen algorithm and equip protocol participants with necessary keys. Note that while we consider the operator as a central trusted entity who generates the cryptographic keys, in practice, a secure multi-party sampling protocol would replace the zkSNARK circuit's trusted setup [10].

Adversarial model. We assume a strong software adversary, who, on the prover, exploits a severe software vulnerability to mount control-flow attacks to divert the attested program's execution. We then assume that the semi-dishonest worker is colluding with the prover in an attempt to convince the verifier that the attested program was executed correctly when in reality, it was not. Finally, we consider an untrusted verifier that attempts to infer details about the prover's program (without colluding with the prover's or worker's adversaries). Note, however, that if the verifier also colludes, this is limited to violating the protocol's privacy objective.

Objectives. Our protocol's objectives are threefold: (i) verifiers always reject a proof unless the prover executed the expected program (or segment thereof) correctly and no control-flow attack was present, (ii) any attempt by the worker to manipulate inputs during proof generation results in a rejection, and (iii) verifiers neither require nor learn any program details from the verification process.

5 THE ZEKRA PROTOCOL

CFG conformance. For the worker to convince an untrusted verifier that an execution path $\mathcal{EP}$ is benign according to the reference program's CFG in zero-knowledge, she must prove the statement "I have successfully verified that $\overline{\mathcal{EP}}$ is a legal path in $\overline{CFG}$, which began at node $n_\triangleright$ and ended at node $n_\blacktriangleleft$, where $\overline{CFG}$ is the preimage of h_1". To prove this statement, we embed it in a circuit C, which we refer to as the ZEKRA circuit, where $\overline{\textbf{overlined}}$ variables denote secret inputs to the circuit as described in Section 3.3, i.e., we have the secret $\overline{u} = \{\overline{\mathcal{EP}}, \overline{CFG}\}$ and public $x = \{n_\triangleright, n_\blacktriangleleft, h_1\}$ inputs.

To allow verifiers to verify whether the correct program's CFG was considered for a given proof, we assume verifiers know the digest of the program's CFG as a reference value, $h_1 = \mathsf{H}(CFG||r_1^\lambda)$, where r_1 is some sufficiently-long random padding (blinding factor) added to the CFG preimage to protect against hammering and linkage attacks and λ denotes the security parameter. (In our case, we consider $\lambda = 254$, corresponding to a 254-bit prime field $\mathbb{F}_p$.) Thus, given a valid proof π over C and public inputs x used in the proof generation, verifiers can verify that the correct CFG was considered by checking that the public input digest matches the expected reference value. However, verifiers cannot infer anything about the CFG preimage, which was supplied as a secret input.

Similarly, to let verifiers determine whether the secret execution path supplied (attested) by the prover also connects the expected CFG nodes, e.g., that it entered as expected at node n_1 and exited at node n_9 in Fig. 1 (thus marking a successful execution), we grant verifiers knowledge about the CFG's contextually relevant entry node $n_\triangleright$ and exit node $n_\blacktriangleleft$, respectively. These nodes are public inputs to the ZEKRA circuit to let verifiers observe them and are used internally to verify the start and end of the supplied execution path. To simplify the discussion, we assume that each CFG has a unique entry node, $n_\triangleright$, and a unique exit node, $n_\blacktriangleleft$. However, the procedure is the same regardless of the considered granularity (e.g., program level or function level), where a CFG might have multiple legal entries or exit nodes. Note that since the interlinking execution path $n_\triangleright \rightsquigarrow n_\blacktriangleleft$ remains secret, the verifier cannot infer anything about the execution path from observing the endpoints.

Furthermore, note that here $n_\triangleright$ and $n_\blacktriangleleft$ do not refer to the actual memory addresses of the corresponding BBLs in the program CFG but to numeric labels that have been assigned to the corresponding nodes in the CFG. Specifically, because we must traverse the CFG in the ZEKRA circuit, we must represent the CFG as a traversable data structure, and using the nodes themselves to index the structure allows for more optimized lookups. (We discuss how we represent the CFG in Section 5.1.) However, since the prover will record and attest to the raw execution path, which includes the actual memory addresses, we assume a mapping $\mathcal{M}$ to help the ZEKRA circuit first translate the recorded addresses into their corresponding numerical label representation. The mapping is simply a list of the possible memory addresses (i.e., nodes in the program CFG), where the index of an address denotes its numeric label.[1] Then, similar to the CFG, the circuit accepts $\mathcal{M}$ as secret input and $h_3 = \mathsf{H}(\mathcal{M}||r_3^\lambda)$ as public input, where r_3 is the random padding (blinding factor) added to $\mathcal{M}$'s preimage, which the verifier (who knows only $\mathsf{H}(\mathcal{M}||r_3^\lambda)$ as a reference value) can verify by observing the circuit's public inputs.

Path authenticity. Note that anyone knowledgeable about the program or its CFG can identify paths that will satisfy the ZEKRA circuit. Thus, to convince verifiers that the secret execution path for a particular proof was recorded on the prover and not produced by someone else, we assume that each prover's trusted tracer has a certified asymmetric key pair $\{\mathsf{tpk}, \mathsf{tsk}\}$, where tpk denotes the public key, and tsk denotes the secret key, respectively. It follows that verifiers must know a prover's public key to verify the authenticity of attestation materials signed using that prover's secret key.

One method of convincing the verifier about the path's authenticity is requiring a prover to sign the recorded execution path $\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{H}(\mathcal{EP}), \mathsf{tsk})$, have the circuit accept Sig and $\mathsf{H}(\mathcal{EP})$ as secret inputs and tpk as a public input, and then have the circuit use tpk to verify internally that Sig is a valid signature over $\mathsf{H}(\mathcal{EP})$ and $\mathsf{H}(\mathcal{EP})$ is the correct digest of $\mathcal{EP}$. Then the verifier can verify that the correct prover authenticated the execution path by checking if the correct tpk was supplied. However, the problem is that signature verification is expensive in terms of circuit size since most algebraic signature schemes are not compactly expressed over a field $\mathbb{F}_p$. For example, expressing the RSA algorithm, which heavily relies on modular exponentiation and long integer arithmetic, yields close to 90K constraints [33], even considering a hardcoded modulus and considerable optimizations. While there exist techniques to reduce the complexity, e.g., by using a small public key exponent [39], there are currently, to the best of our knowledge, no efficient general-purpose signature schemes for circuits.

Another method is to have the prover prove possession of the secret key behind its public key. For example, assuming that the RSA cryptosystem is considered, we could include the substatement "I know $\overline{p}$ and $\overline{q}$, where $\overline{p} \times \overline{q} = n$" as part of the ZEKRA circuit's underpinning statement since knowledge of $\overline{p}$ and $\overline{q}$ for some public key modulus n proves possession of the secret key. However, this would require the prover herself to generate the proof, which is unsatisfactory, especially since CFA schemes generally consider resource-constrained or heavily embedded devices and have the prover only be concerned with tracing the program before outsourcing the signed execution path to the verifier. Therefore, without complicating the prover, we must design the ZEKRA circuit with the intention of the proof generation being outsourced to workers.

For the third method of proving path authenticity, which we opted for in our current version, signature verification is performed outside the circuit, as inspired by [39]. The idea is for the circuit to accept $h_2 = \mathsf{H}(\mathcal{EP}||nce^\lambda||r_2^\lambda)$ as public input, where nce^λ is a fresh nonce generated by the verifier to ensure freshness, and r_2^λ is some random padding (blinding factor) generated and added to the execution path by the prover. The nonce is given as public input to the circuit to let verifiers ascertain freshness while the blinding factor is kept secret. The circuit then verifies internally that the secret execution path $\overline{\mathcal{EP}}$, padded with the nonce and blinding factor, is indeed the correct preimage of h_2. As in the first method, the prover also signs the recorded execution path $\mathsf{Sig} \leftarrow \mathsf{Sign}(\mathsf{H}(\mathcal{EP}||nce^\lambda||r_2^\lambda), \mathsf{tsk})$. The worker's proof and prover's signature are then given to the verifier, who verifies that the public digest h_2 used in the proof generation matches the prover's signed digest.

We give more details on the ZEKRA circuit in Section 6. Let us first bring it all together and clarify the overall protocol.

The protocol. Fig. 2 shows a prover and a verifier engaging in the protocol. To ensure freshness, the verifier challenges the prover with a nonce *nce* and a reference @P to the program to be executed and attested. In practice, the attested region is only a subset of the entire program [3, 49], e.g., a security-critical function or code section. The prover then executes the program while its trusted tracer chronologically traces the executed path $\mathcal{EP}$ when executing the region to be attested. Once the program concludes, the execution path is first hashed together with the verifier's nonce and some freshly sampled blinding factor and then signed. The signed digest and secret ingredients are then given to the worker, who, before generating a proof over the ZEKRA circuit, first converts the received execution path into its numerical label representation $\mathcal{L}$ using the address-to-label mapping $\mathcal{M}$, which the circuit can then verify to be done correctly instead of having to perform the computationally-intensive conversion task (further optimizations are discussed in Section 6). The worker then computes a zkSNARK proof by executing Prove and passing in as secret inputs: the attested program's reference materials (i.e., the CFG and address-to-label mapping $\mathcal{M}$, along with their blinding factors), the attested execution path (including its blinding factor), and the numerical representation of the attested execution path $\mathcal{L}$. As public input, the worker passes in the digests of the CFG, mapping $\mathcal{M}$, and the execution path, together with the relevant entry and exit nodes and the verifier nonce.

Finally, the proof, its public inputs, and the prover's signature over the execution path commitment digest are given to the verifier, who is convinced that the intended program was executed correctly, in the absence of any control-flow attacks, on the prover if: (i) the proof is satisfied under the circuit's verification key, (ii) the execution path commitment supplied as public input to the proof generation–whose corresponding preimage ingredients were supplied as secret inputs and verified internally in the circuit to hash to the public commitment digest–was signed by the prover, (iii) the CFG and mapping $\mathcal{M}$ of the intended program were considered, and (iv) the correct start and end nodes were visited. If these criteria are satisfied, the verifier is convinced that the intended program (or segment thereof) was executed correctly on the prover.

5.1 Building Blocks

Before explaining our circuit's design, we must understand how we represent and work with execution paths and program CFGs.

Tracing. When chronologically tracing a program's execution, we assume that each execution path $\mathcal{EP}$ (of some length E) is marshalled in the form of a sequence of control-flow transitions: $\mathcal{EP} = ($

[1]Another benefit of keeping the CFG representation abstract inside the circuit is that we are not limiting what is being mapped. For example, instead of mapping BBL addresses, we could also include the hashes of the executed code as done in ATRIUM [53]. We show how this extension has negligible performance impact in Section 7.2.

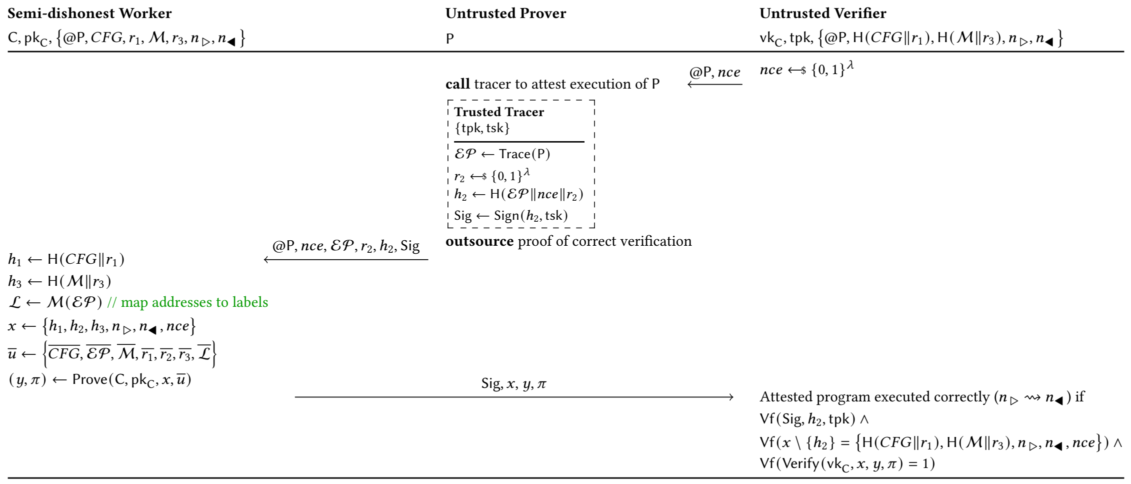

Semi-dishonest Worker
$\mathsf{C}, \mathsf{pk_C}, \{@\mathsf{P}, CFG, r_1, \mathcal{M}, r_3, n_{\triangleright}, n_{\blacktriangleleft}\}$
Untrusted Prover
P
Untrusted Verifier
$\mathsf{vk_C}, \mathsf{tpk}, \{@\mathsf{P}, \mathsf{H}(CFG\|r_1), \mathsf{H}(\mathcal{M}\|r_3), n_{\triangleright}, n_{\blacktriangleleft}\}$
$nce \leftarrow\!\!\$\ \{0,1\}^{\lambda}$
$@\mathsf{P}, nce$
call tracer to attest execution of P
Trusted Tracer
$\{\mathsf{tpk}, \mathsf{tsk}\}$
$\mathcal{EP} \leftarrow \mathsf{Trace}(\mathsf{P})$
$r_2 \leftarrow\!\!\$\ \{0,1\}^{\lambda}$
$h_2 \leftarrow \mathsf{H}(\mathcal{EP}\|nce\|r_2)$
$\mathsf{Sig} \leftarrow \mathsf{Sign}(h_2, \mathsf{tsk})$
outsource proof of correct verification
$@\mathsf{P}, nce, \mathcal{EP}, r_2, h_2, \mathsf{Sig}$
$h_1 \leftarrow \mathsf{H}(CFG\|r_1)$
$h_3 \leftarrow \mathsf{H}(\mathcal{M}\|r_3)$
$\mathcal{L} \leftarrow \mathcal{M}(\mathcal{EP})$ // map addresses to labels
$x \leftarrow \{h_1, h_2, h_3, n_{\triangleright}, n_{\blacktriangleleft}, nce\}$
$\overline{u} \leftarrow \{\overline{CFG}, \overline{\mathcal{EP}}, \overline{\mathcal{M}}, \overline{r_1}, \overline{r_2}, \overline{r_3}, \overline{\mathcal{L}}\}$
$(y, \pi) \leftarrow \mathsf{Prove}(\mathsf{C}, \mathsf{pk_C}, x, \overline{u})$
Sig, x, y, π
Attested program executed correctly $(n_{\triangleright} \rightsquigarrow n_{\blacktriangleleft})$ if
$\mathsf{Vf}(\mathsf{Sig}, h_2, \mathsf{tpk}) \wedge$
$\mathsf{Vf}(x \setminus \{h_2\} = \{\mathsf{H}(CFG\|r_1), \mathsf{H}(\mathcal{M}\|r_3), n_{\triangleright}, n_{\blacktriangleleft}, nce\}) \wedge$
$\mathsf{Vf}(\mathsf{Verify}(\mathsf{vk_C}, x, y, \pi) = 1)$

Figure 2: The ZEKRA protocol, where, upon request, a prover attests to the execution of a program before outsourcing the task of convincing the untrusted verifier about the execution's correctness in zero-knowledge to a semi-dishonest worker.

$t_1, t_2, \ldots, t_n)$, where each transition $t_i = (jmpkind, n_{dst}, n_{ret})$ includes a 2-bit identifier for the type of transition, i.e., whether the transition was caused by a jump, function call, or function return, the destination addresss (supposed entry of the target BBL), and a return node n_{ret}, which for calls points to a BBL where the callee function should return. Note that knowing the transition type enables shadow stack emulation for ensuring back edge integrity.

Control-Flow Graph Representation. A core part of the ZEKRA circuit is how we represent and traverse a CFG. To model the set of legal transitions, we can either encode the set of legal edges as an adjacency matrix or adjacency list. In the matrix format, we can represent a digraph $G = (N, E)$ as a two-dimensional array M of size $N \times N$, where a slot $M(n_i)(n_j) = 1$ indicates that an edge exists from node n_i to node n_j. The advantage of the matrix is that we can determine in $O1$ whether an edge exists from n_i to n_j. However, since the matrix has space complexity of $O|N|^2$ it requires a prohibitively large data structure to be expressed in an arithmetic circuit. Contrarily, in the adjacency list, we only store a node's reachable neighbors, which reduces the space complexity to $O|N| + |E|$ but increases query time complexity to $O|N|$. While the space complexity is better than the matrix, it can become expensive for dense areas in a CFG where a node might have many reachable neighbors (e.g., a program switch with a large jump table).

To reduce the space complexity even further, we leverage the idea behind the **IndexedBitArrayEdges** representation as proposed in [36], which takes advantage of the concentration of edges in specific areas of the adjacency matrix. With this encoded representation we use a single byte to represent eight possible out-neighbors. Using an array, we construct a data structure of $(bucket + 8)$-bit elements, one for each neighbor label interval with the same quotient when divided by 8. Each element's first $bucket - 8$ bits represents the quotient (bucket), while the last byte serves as a set of 8 flags indicating whether each possible edge exists in this interval. Note that it follows that $bucket$ must at minimum be $\lfloor \log_2((N-1)/8) \rfloor + 1$ bits for us to represent all possible quotients of CFGs with N nodes.

Since the circuit performs execution path verification on CFGs with all nodes relabeled using consecutive integers (where the relabeling is reflected in the mapping $\mathcal{M}$), we can represent any N-node abstract CFG as a single-dimensional adjacency list of size N, whose indices correspond to CFG nodes and contain the indexed node's encoded neighbors. Also, since the maximum label is $N - 1$ when numbered from 0, the circuit only needs to allocate $\lfloor \log_2(N - 1) \rfloor + 1$ bits per label to represent the execution path.

To better understand how we apply the encoding, assume that a node has the following set of neighbors: $\{288, 289, 290, 291, 292, 293, 294, 614\}$. We can group the neighbors in two sets: $\{288, 289, 290, 291, 292, 293, 294\}$, $\{614\}$, where the first set shares bucket 36 when divided by 8, and the second set share the bucket 76. We then iteratively store the bucket of each neighbor set as the first $\lfloor \log_2((N - 1)/8) \rfloor + 1$ bits and the remainders ($rems$) as the neighboring byte. We refer to each such $(bucket, rems)$ pair as a level and use ℓ to denote the maximum levels of the encoded adjacency list. Note, however, that the number of levels needed depends on how the adjacency list's labels are arranged. For example, in our example, we require two levels to accurately represent all eight neighbors, where the 0th to the 6th bit of the first level's $rems$ are set to 1 to indicate the first seven neighbors. However, if we could rearrange the numerical adjacency list such that the considered node's neighbors all shared the same quotient, it would only need one level. (We defer this graph optimization problem and other CFG reduction/compression methods as they complement our work.)

Finally, given an encoded adjacency list $\mathcal{AL}$, we determine if node n_j is a valid neighbor of node n_i by verifying that there exists some $(bucket, rems)$ pair in $\mathcal{AL}$, such that $\lfloor n_j/8 \rfloor = bucket \wedge rems[n_j \bmod 8] = 1$, where $rems[n_j \bmod 8]$ denotes a bit in $rems$ at position $n_j \bmod 8$. In other words, we check that the destination node's bucket exists and the corresponding remainder bit is on.

$\mathcal{AL}$, $\mathcal{EP}$ and $\mathcal{M}$ hashing. When selecting a suitable hashing function H to utilize in our circuit, the deciding factor is how inexpensively it can be expressed in an arithmetic circuit. Fortunately, several hash functions have been proposed due to increased attention to circuit-based zero-knowledge proofs. The most recent, which currently offers the best performance, is the cryptographic permutation function called Poseidon [22], which takes a set of elements of a certain field $\mathbb{F}$, called scalars, as inputs and outputs one scalar. The number of inputs determines the width $w = r + c$ of the internal state, where r and c are called the *rate* and *capacity* of the permutation function. Setting the capacity to one field element in a 254-bit field $\mathbb{F}$ offers a 128-bit security level and using a rate (arity) of 4 essentially corresponds to a 4-to-1 compression function.

In our case, we configured Poseidon-128 with a width of 9 to achieve a rate of eight field elements per call, which allows us to walk over the data structures during hashing more quickly. Since the considered data structures can be arbitrarily/selectively large, we used the proposed Poseidon constant-length sponge-based construction [22]. Let $X = (x_1, \ldots, x_m)$ refer to an execution path $\mathcal{EP} = (t_1, \ldots, t_m)$ of l-bit transitions, an adjacency list $\mathcal{AL} = (e_1, \ldots, e_m)$ of l-bit encoded neighbor entries, or a mapping $\mathcal{M} = (a_1, \ldots, a_m)$ of l-bit entries, respectively. We then hash X into a single scalar (i.e., field element) as follows:

(1) Compress X by successively fitting $\lfloor(\mathbb{F}\text{'s bitwidth})/l\rfloor$ l-bit values from X into one field element and storing the resulting value in X'. Let t denote the size of X'.
(2) Pad X' with zero elements up to the multiple of 8, then split it into chunks $w_1, w_2, \ldots, w_{\lceil t/8 \rceil}$.
(3) Apply the permutation function Poseidon to the capacity element and the first chunk.

$$(h_1^1, h_1^2, \ldots, h_1^9) \leftarrow \textsc{Poseidon}(len \times 2^{64} + (o-1), w_1)$$

(Note that the capacity field is set to $len \times 2^{64} + (o-1)$, where len is the input length and o is the output length (usually $o = 1$). In our case the input length is 8 field elements, and the output length is 1 field element.)
(4) Until no more chunks are left, apply the permutation:

$$(h_i^1, h_i^2, \ldots, h_i^9) \leftarrow \textsc{Poseidon}(h_{i-1}^1, h_{i-1}^2 + w_i^1, \ldots, h_{i-1}^9 + w_i^8)$$

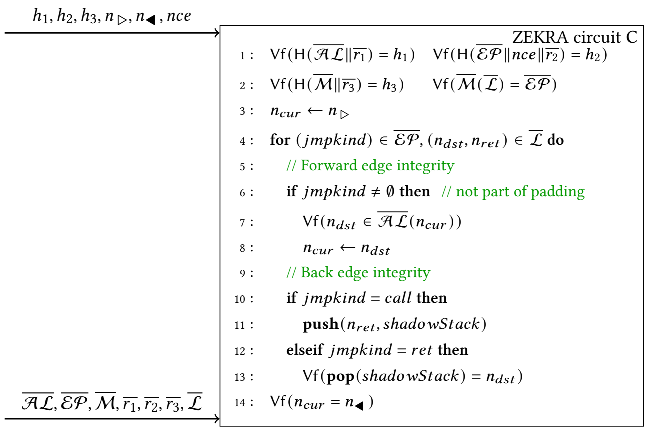
(5) Output o output elements from the rate part of the state, i.e., in our case, the digest of X is the second element:

$$\mathsf{H}(X) = h_{\lceil t/8 \rceil}(2)$$

We utilize this sponge construction in our circuit to verify the considered secret preimages against their corresponding public digests before relying on them to accurately report on the expected program's reference materials and the execution path. Thus, the public digests must also be computed similarly on the outside.

6 ON THE DESIGN OF THE ZEKRA CIRCUIT

Before detailing our design choices, we briefly overview the different circuit components/gadgets. The high-level algorithm of the ZEKRA circuit is presented in Fig. 3 along with its secret and public inputs. Note that we have replaced the CFG in Fig. 2 with the encoded adjacency list $\mathcal{AL}$ of size N. After verifying the preimages and the correctness of the translation of the attested execution path $\mathcal{EP}$ of size E into its label representation $\mathcal{L}$, the circuit traverses the label version of the execution path to verify its legality, according to the adjacency list and the expected entry and exit nodes.


$h_1, h_2, h_3, n_{\triangleright}, n_{\blacktriangleleft}, nce$
$\overline{\mathcal{AL}}, \overline{\mathcal{EP}}, \overline{\mathcal{M}}, \overline{r_1}, \overline{r_2}, \overline{r_3}, \overline{\mathcal{L}}$
ZEKRA circuit C

```
$\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{AL}} \| \overline{r_1}) = h_1)$   $\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{EP}} \| nce \| \overline{r_2}) = h_2)$
$\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{M}} \| \overline{r_3}) = h_3)$   $\mathsf{Vf}(\overline{\mathcal{M}}(\overline{\mathcal{L}}) = \overline{\mathcal{EP}})$
$n_{cur} \leftarrow n_{\triangleright}$
for $(jmpkind) \in \overline{\mathcal{EP}}, (n_{dst}, n_{ret}) \in \overline{\mathcal{L}}$ do
  // Forward edge integrity
  if $jmpkind \neq \emptyset$ then  // not part of padding
    $\mathsf{Vf}(n_{dst} \in \overline{\mathcal{AL}}(n_{cur}))$
    $n_{cur} \leftarrow n_{dst}$
  // Back edge integrity
  if $jmpkind = call$ then
    push$(n_{ret}, shadowStack)$
  elseif $jmpkind = ret$ then
    $\mathsf{Vf}(\text{pop}(shadowStack) = n_{dst})$
$\mathsf{Vf}(n_{cur} = n_{\blacktriangleleft})$
```

Figure 3: High-level algorithm of the outsourceable circuit.

Forward edge integrity. To verify that the execution path satisfies forward edge integrity, we maintain a state variable n_{cur} (initialized to $n_{\triangleright}$) as we walk the execution path to verify its legality, i.e., that it is a continuous sequence of transitions that only flow through neighboring (adjacent) nodes, as follows. For each transition t_i in the execution path, we consult n_{cur}'s neighbors from the adjacency list and verify that t_i's destination node n_{dst} is indeed listed as a valid neighbor, and if so, we update n_{cur} to n_{dst}. Thus, by verifying that it reached the final node $n_{\blacktriangleleft}$ on exit, we are certain, by transitivity, that the execution path is legal and also correctly connects the expected endpoints. However, note that while a node can have several reachable and equally valid neighbors in the forward direction, this is not true for backward edges. Specifically, when a function returns, *it should only return to where the caller intended.*
Back edge integrity. Therefore, to ensure exact back edge integrity, we consider, similar to other CFA schemes [49, 50], the use of a shadow stack (of some depth D) to simulate the traditional program stack as we walk the execution path. For function calls, we push the return node n_{ret} on the stack, and for returns, we verify that n_{dst} indeed is the stack's topmost element.

6.1 Circuit Design Challenges

Besides the challenges already described, e.g., authenticating execution paths in Section 5, representing the CFG as a space-efficient encoded adjacency list, and selecting a suitable hashing method in Section 5.1, we are missing the actual execution path traversal.

Due to the complexity of expressing computations as circuits, many circuit construction tools include programmable interfaces and compilers to optimize the translation of computations expressed in a higher-level language into circuits [17, 29, 33]. However, while the circuit compilers let us not worry about the low-level wiring process, they are not as mature as standard program compilers, and there are thus many employable techniques to further reduce the number of constraints needed to express a particular program. Specifically, note that the complexity of any program in terms of the number of constraints it compiles down to is the sum of the cost of expressing all statements, all loop iterations, and accounting for all branches (see Appendix B for more details). Therefore, operations that cannot easily be expressed, e.g., modulo, division,

exponentiation, "bit twiddling", and random-memory access, are *costly*. Fortunately, as stated in [33], an essential strategy to optimize complexity is the observation that it usually suffices and is generally cheaper to have a circuit verify a computation's correctness instead of computing the function in the forward direction. For example, $y = x/a$ can be verified more efficiently by checking that $ya = x$ rather than computing the division in the forward direction. Similarly, when working with modular arithmetic, the joint statements $r \equiv x \bmod q$, $y = \lfloor x/q \rfloor$, where q is the modulus, can be verified more efficiently by checking that $yq + r = x \wedge r < q$.
Random-accessed memory. While sequentially walking the execution path incurs a negligible cost, verifying each transition's destination node n_{dst} according to the n_{cur}th entry in $\mathcal{AL}$ requires random memory access, which is expensive since no information is known about which element is being accessed during compile-time. The same applies to verifying that the numeric execution path $\mathcal{L}$ is a correct translation of $\mathcal{EP}$ according to $\mathcal{M}$, as shown in (1).

$$\mathsf{Vf}(\forall i \in \{0 \ldots E\} : \mathcal{M}(\mathcal{L}(i)(dst)) = \mathcal{EP}(i)(dst) \wedge \mathcal{M}(\mathcal{L}(i)(ret)) = \mathcal{EP}(i)(ret)) \quad (1)$$

The typical, naive approach for realizing dynamic memory is performing a linear scan of the entire array memory to select one element for each memory access, which results in Okn cost for making k memory accesses where n is the array's total memory size. Another approach, supported by recent compilers [33, 51], involves a permutation network with complexity $O(n + k)(\log(n + k))$. However, seeing how the smart memory recently proposed in xJsnark [33] has outperformed the permutation network with a complexity of $Ok\sqrt{n}$, promising $2\sqrt{n} + \log_2 n$ constraints per access, we opted to express the current version of ZEKRA in xJsnark.
Representing $\mathcal{AL}$. Note that xJsnark currently only supports one-dimensional arrays with its smart memory type. Therefore, we have the circuit accept the adjacency list $\mathcal{AL} = (neighbors_1, \ldots, neighbors_m)$ as a single, node label indexable array of field elements, whose sizes correspond to the size of our finite prime field $\mathbb{F}_p$ (in our case 254 bits), i.e., each node n_i's $(bucket, rems)$ pairs are consecutively concatenated into one 254-bit field element $neighbors_i = p_1||p_2|| \ldots ||p_\ell$. However, the challenge with the concatenation is determining whether a specific destination node n_{dst} is a neighbor of some node n_i when $neighbors_i$ is given as a numeric value.

While we could enforce a specific type on $\mathcal{AL}$'s elements, e.g., that they be unsigned integers instead of native field elements, allowing us to iterate over the $(bucket, rems)$ pairs using either bitwise or division/exponentiation operations, recall that these operations are expensive. Furthermore, by restricting input values to n-bit unsigned integers, the circuit must verify that the supplied values are n bits while continuously ensuring that any operation on the integers results in a value that fits within that range, essentially resulting in $n + 2$ additional constraints for each integer [33]. Contrarily, by keeping the input values as native field elements, they are guaranteed to remain within a certain bitwidth.

Therefore, instead of retrieving each $(bucket, rems)$ in the circuit, we apply the power of SNARK verification by accepting (as secret input) another two-dimensional (non-smart) array $\mathcal{AL}'$ of size $E \times 2\ell$, containing the pairs already split into separate elements. However, contrary to $\mathcal{AL}$, we access $\mathcal{AL}'$ sequentially, which is made possible by requiring $\mathcal{AL}'$ to be ordered with relevance to the transitions, i.e., the ith index of $\mathcal{AL}'$ contains n_{cur}'s pre-split $(bucket, rems)$ pairs at transition i, which is easily arranged by the worker. However, before relying on $\mathcal{AL}'(i)$, the circuit first verifies that $\mathcal{AL}'(i)$ correctly represents $\mathcal{AL}(n_{cur})$ by checking that $\mathcal{AL}'(i)$ computes to $\mathcal{AL}(n_{cur})$ as shown in (2).

$$\mathsf{Vf}\Big(\Big(\sum_{j=0,2,\ldots}^{2\ell-1} \mathcal{AL}'(i)(j+1) \times 2^{\lfloor j/2 \rfloor (bucket\ \text{bitwidth}+8)+8} + \mathcal{AL}'(i)(j) \times 2^{\lfloor j/2 \rfloor (bucket\ \text{bitwidth}+8)}\Big) = \mathcal{AL}(n_{cur})\Big) \quad (2)$$

Due to space limitations, we detail how we apply further optimizations to efficiently check whether a particular neighbor exists in Appendix C. Further, Appendix D summarizes the final design.

7 EMPIRICAL PERFORMANCE EVALUATION

Our evaluation addresses the questions of: (i) how efficient is ZEKRA for different program complexities and (ii) how tolerable are the combined costs for CFA of real-world *deeply embedded applications*.

7.1 Asymptotic Performance

Table 1 shows the complexity of our design considering a 254-bit field $\mathbb{F}$ and a Poseidon-128 implementation with an arity/rate of 8 field elements and a cost of $\mathfrak{C} = 405$ constraints per call. For comparison, we also show the complexity *without* the space efficient adjacency list encoding described in Section 5.1, i.e., where each entry in $\mathcal{AL}$ is simply the concatenation of that node's neighbors, where Δ denotes the maximum supported neighbors of any node.

The first four rows in Table 1 give the complexity of verifying: the adjacency list (with and without encoding), attested execution path, and mapping, respectively. Note that it takes a single constraint to verify that a computed digest matches its corresponding public reference. Thus, we only consider the complexity of the hashing. Note here that the bit-space needed when compressing the unencoded adjacency list into the least number of field elements (to minimize the number of calls to Poseidon) is directly affected by the number of supported neighbors Δ (second row). In contrast, the bit-space needed for the encoded adjacency list is affected by the number of levels (ℓ) used for the encoding (first row). To illustrate the power of the encoding, note that we can only store 25 10-bit labels in a 254-bit field element. Thus, since we only have a one-dimensional adjacency list of field elements, we can only represent adjacency lists with $\Delta = 25$. However, using the encoding, we can store a total of 16 levels ℓ of 15-bit $(bucket, rems)$ pairs (we need 7-bit buckets when considering 10-bit labels), which can hold 128 labels. Thus, using the encoding, we can reduce the number of calls to Poseidon and also represent more complex adjacency lists using fewer bits.

Note that verifying the correctness of the worker's translation of an execution path of length E requires $2E$ accesses to $\mathcal{M}$ inside the circuit since we must verify each transition's destination address and (possible) return address. This complexity is shown in row five of Table 1, which evidently dominates the overall circuit complexity.

The complexity of verifying that each transition's destination node is valid according to an encoded or unencoded adjacency list is shown on rows six and seven of Table 1, respectively. Note that we accept the pre-split version of $\mathcal{AL}$ as a sequentially accessed, two-dimensional structure $\mathcal{AL}'$ in both cases. For the encoded version, $\mathcal{AL}'$ contains the $(bucket, rems)$ pairs as described in Section 6.1.

Table 1: Component complexity in terms of the number of constraints when compiled using xJsnark, where $\mathfrak{C} = 405$ is the cost per call to Poseidon. The table also shows the cost if we store digests in $\mathcal{AL}$ to emulate more space (beyond $\mathbb{F}$'s bitwidth).

Circuit Component/Gadget	Complexity	Actual (Asymptotic) Total Cost	Using Poseidon digests as $\mathcal{AL}$ elements
C1: $\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{AL}} \Vert \overline{r_1}) = h_1)$	$O(N)$	$\mathfrak{C}\lceil (N\ell(bucket\text{'s bitwidth} + 8) + 1)/(8\mathbb{F}\text{'s bitwidth})\rceil$	$\mathfrak{C}\lceil N/8 \rceil$
- (wo. $\mathcal{AL}$ encoding)	$O(N)$	$\mathfrak{C}\lceil (N\Delta label\text{'s bitwidth} + 1)/(8\mathbb{F}\text{'s bitwidth})\rceil$	$\mathfrak{C}\lceil N/8 \rceil$
C2: $\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{EP}} \Vert \overline{nce} \Vert \overline{r_2}) = h_2)$	$O(E)$	$\mathfrak{C}\lceil (E2addr\text{'s bitwidth} + 4)/(8\mathbb{F}\text{'s bitwidth})\rceil$	N/A
C3: $\mathsf{Vf}(\mathsf{H}(\overline{\mathcal{M}} \Vert \overline{r_3}) = h_3)$	$O(N)$	$\mathfrak{C}\lceil (Naddr\text{'s bitwidth} + 1)/(8\mathbb{F}\text{'s bitwidth})\rceil$	N/A
C4: $\mathsf{Vf}(\overline{\mathcal{M}}(\overline{\mathcal{EP}}) = \overline{\mathcal{L}})$	$O(E\sqrt{N})$	$2E(2\sqrt{N} + \log_2 N) + 10E$	N/A
C5: Forward edge integrity	$O(E\sqrt{N})$	$E(2\sqrt{N} + \log_2 N) + E(\ell + 38 + label\text{'s bitwidth})$	$+E\mathfrak{C}\lceil (\ell(bucket\text{'s bitwidth} + 8))/(8\mathbb{F}\text{'s bitwidth})\rceil$
- (wo. $\mathcal{AL}$ encoding)	$O(E\sqrt{N})$	$E(2\sqrt{N} + \log_2 N) + E(\Delta + 11 + label\text{'s bitwidth})$	$+E\mathfrak{C}\lceil (\Delta label\text{'s bitwidth})/(8\mathbb{F}\text{'s bitwidth})\rceil$
C6: Backward edge integrity	$O(E\sqrt{D})$	$2E(2\sqrt{D} + \log_2 D) + E(28 + 2\log_2 D)$	N/A

For the unencoded version, $\mathcal{AL}'$ contains the individual neighbors. In both cases, we maintain our state variable (n_{cur} in Fig. 3) as an unsigned integer, which is used to access $\mathcal{AL}$ and whose bitwidth is determined by the maximum label. Furthermore, in both cases, we perform a linear search over $\mathcal{AL}'$ to find a match, which requires either ℓ (using a step size of 2) or Δ iterations with and without the encoding. (Note that Δ quickly outgrows ℓ.) Finally, while negligible, note that the slightly higher (constant) cost per transition in the case of the encoded version is the cost of our proposed method of verifying a linear system of equations as described in Appendix C to check if a destination node exists in a $(bucket, rems)$ pair.

Recall that a verifiable program's complexity in terms of the number of constraints it compiles down to is the sum of the cost of all branches (see Section 6.1 and Appendix B). Thus, the complexity of the back-edge integrity component (last row) includes the sum of both branches (push and pop) per transition. However, note that we can usually keep the stack depth, D, small (unless attesting to highly nested/recursive code). Hence the double memory cost for this component is less significant than that of the fourth component.

Supporting more neighbors. To store more neighbors in $\mathcal{AL}$, we need more space per node element. Without compiler support for two-dimensional RAM, a naive approach is to emulate it with more arrays. However, in this case, the memory access cost grows proportionally to the number of arrays, i.e., for two parallel arrays, the cost per transition becomes $2 \times (2\sqrt{N} + \log_2 N)$. Another approach, whose cost is also shown in Table 1, is to instead store the hash of a node's neighbors as a field element in $\mathcal{AL}$ whose preimage is then given as a two-dimensional array in $\mathcal{AL}'$. Then, for each transition i we simply verify that $\mathsf{H}(\mathcal{AL}'(i)) = \mathcal{AL}(n_{cur})$ before performing a neighbor lookup in $\mathcal{AL}'(i)$, where $\mathcal{AL}'(i)$ now supports an arbitrarily large neighbor space. Note, however, that this choice comes at a cost proportional to the number of Poseidon calls we need to make per transition to perform the verification and thus is only mentioned as an alternative method for our approach to scale in support of attesting to arbitrary CFG complexities. Specifically, to get 8 field elements (the considered arity) of neighbor storage per node (allowing for ≈ 1024 neighbors using the encoding when considering 10-bit labels), this comes at the cost of one Poseidon call per transition, i.e., giving an overall (additional) cost of $E \times \mathfrak{C}$.

7.2 Empirical Performance

Datasets. Table 2 shows some extracted datasets for a selection of demonstrative applications taken from the embench-iot suite [18], which comprises a set of real-world, deeply embedded applications[2]. To ensure reproducibility, we coded helpers [14] to perform all evaluation steps. For compilation, we use GCC options `-Os` `-g0` and the `-fno-optimize-sibling-calls` flag for deactivating *sibling and tail recursive calls* optimizations. We then use the angr [48] binary analysis tool for extracting static CFGs and sample execution paths through symbolic execution, where the sample paths simulate paths as recorded by a prover. To generate the trusted reference material, we use the NetworkX Python package [25] for translating the extracted CFGs into isomorphic, numerically labeled representations, which are then converted into corresponding adjacency lists $\mathcal{AL}$ and used to derive the address-to-label mappings $\mathcal{M}$.

Labeling. Note that the minimum number of levels (ℓ) needed to encode a specific adjacency list is determined by the maximum number of quotients (i.e., buckets) shared by any node's neighbors, which depends on the way the nodes are labeled numerically. In our experiments, we labeled each extracted CFG's nodes using consecutive integers in the order they appeared. (We defer the graph optimization problem of finding the most optimal ordering as future work.) Note, however, that even without any special preprocessing, we can already observe in Table 2 for `picojpeg` how "consecutivity" among the neighbors of a node allows us to effectively represent the maximum outdegree $\Delta = 37$ using only $\ell = 7$ levels, meaning we need only 105 bits to encode each node's neighbors (each level is 15 bits). Without the encoding, we would need 370 bits to represent 37 neighbors, which already exceeds the considered prime field $\mathbb{F}$.

Compression. As noted in [2] and utilized in most CFA works, the most basic method of reducing the path explosion problem without loss of accuracy is to prune repetitions in the execution path since they do not affect the *legality* of the *control-flow*. Similarly, we consider that recorded execution paths are compressed such that each unique loop path only occurs once, i.e., all consecutively repeating sequences are discarded. Note that this compression *only removes duplicates* in the path, allowing us to use a smaller circuit for verification. However, the compression does *not* affect our ability to detect control-flow attacks (except those that only affect the number of loop iterations). (We discuss extending our approach to attesting to the number of loop iterations in Section 8.)

Experimental setup. As described in Section 6.1, we implemented our solution using xJsnark [33], a high-level code-to-circuit compilation framework that employs a mix of optimizations to minimize

[2]Note that these applications only served as data points in our performance evaluation and were not selected by their need for control-flow attestation in practice.

Table 2: Sample datasets from the embench-iot suite of real-world embedded applications, each with 24-bit address space.

	Control-Flow Graph G				Sample recorded execution path (through symbolic execution)					
Application	N	Edges	$\Delta(G)$	ℓ	E (pre[a])	E (post[b])	# loops	Avg. loop length	Avg. # of repetitions	D
aha-mont64	114	151	6	2	997	110	109	2.82 ($\sigma \approx 1.55$)	3.01 ($\sigma \approx 1.50$)	5
crc32	88	106	2	2	3,090	24	1	3.00 ($\sigma \approx 0.00$)	1,023 ($\sigma \approx 0.00$)	6
cubic	147	198	8	3	105	10	1	1.00 ($\sigma \approx 0.00$)	96.00 ($\sigma \approx 0.00$)	5
edn	152	195	2	2	4,889	422	16	5.81 ($\sigma \approx 12.34$)	89.19 ($\sigma \approx 98.89$)	5
huffbench	188	284	3	3	9,894	1,155	84	4.39 ($\sigma \approx 4.27$)	53.77 ($\sigma \approx 155.81$)	6
matmult-int	113	143	8	2	37	37	0	0.00 ($\sigma \approx 0.00$)	0.00 ($\sigma \approx 0.00$)	5
md5sum	129	176	4	3	8,399	382	6	54.33 ($\sigma \approx 119.26$)	537.17 ($\sigma \approx 490.16$)	7
minver	176	252	3	3	324	201	21	5.00 ($\sigma \approx 6.20$)	3.10 ($\sigma \approx 2.43$)	5
nbody	113	140	3	2	108	58	4	5.00 ($\sigma \approx 0.00$)	3.50 ($\sigma \approx 1.12$)	6
nettle-aes	156	211	3	3	1,821	199	11	14.00 ($\sigma \approx 15.21$)	34.27 ($\sigma \approx 71.00$)	7
nettle-sha256	173	245	4	3	295	106	10	2.70 ($\sigma \approx 3.16$)	17.60 ($\sigma \approx 19.64$)	6
nsichneu	853	1500	2	2	659	655	1	4.00 ($\sigma \approx 0.00$)	2.00 ($\sigma \approx 0.00$)	4
picojpeg	633	1168	37	7	1,875	335	6	12.50 ($\sigma \approx 8.46$)	31.67 ($\sigma \approx 20.90$)	11
primecount	105	127	3	3	1,742	1,001	66	8.62 ($\sigma \approx 11.69$)	2.36 ($\sigma \approx 0.64$)	5
sglib-combined	728	1084	5	3	8,191	868	105	12.63 ($\sigma \approx 16.62$)	7.90 ($\sigma \approx 20.19$)	8
slre	347	528	6	2	2,217	609	5	13.40 ($\sigma \approx 12.47$)	10.80 ($\sigma \approx 16.12$)	13
st	123	155	3	3	1,143	65	7	1.57 ($\sigma \approx 0.90$)	99.00 ($\sigma \approx 0.00$)	6
statemate	434	657	2	2	256	102	3	1.34 ($\sigma \approx 0.47$)	31.67 ($\sigma \approx 22.16$)	6
tarfind	102	137	3	3	12,070	443	38	147.0 ($\sigma \approx 161.36$)	13.82 ($\sigma \approx 40.87$)	5
ud	133	174	3	2	393	185	25	3.76 ($\sigma \approx 2.20$)	4.60 ($\sigma \approx 7.31$)	5
wikisort	425	645	7	3	4,610	152	7	26.71 ($\sigma \approx 50.41$)	66.57 ($\sigma \approx 84.31$)	7

[a]before and [b]after path compression (removing all consecutively repeated sequences).

circuit complexity. The high-level code is compiled into arithmetic circuits in an extension of the Pinocchio [42] intermediate opcode format, which, using the jsnark interface [32], are translated into the R1CS constraint system and fed into the libsnark [35] backend for instantiating a particular zkSNARK proof system over the circuit. In our case, we considered libsnark's implementation of the state-of-the-art Groth16 [24] proof system (over the BN128 curve), whose proof is 1016 bits and contains 3 group elements (2 $\mathbb{G}_1$ elements and 1 $\mathbb{G}_2$ element), and 3 pairings dominate verification. We then benchmarked our prototype using libsnark's built-in profiler, which includes the generation of the circuit's proving and verification keys and execution of the proof generation and verification algorithms on our experimental inputs, which were formatted from our real-world extracted datasets. As the worker, we considered a machine with an AMD Ryzen 7 3700X processor and 16 GB of memory (experiments were conducted in a WSL2 environment).

Benchmarks. Table 3 shows benchmarks for generating proofs over some demonstrative circuits (proof verification is constant due to Groth16's underpinnings). Note that when compiling a particular circuit, we must define the sizes of the data structures we want to express. However, even if we fix both E and N to 1K, we can still supply execution paths and adjacency lists $\leq$ 1K by employing padding as described in Appendix D. Thus, the larger circuits in Table 3 (after the fourth row) support all datasets shown in Table 2. Furthermore, note that the reported running times assume that the proving key is preloaded in memory, which holds when proof generation is performed by dedicated workers that expect the key to be used regularly and thus retain it in memory. Further, note that we only list the proving key sizes since Groth16's verification keys have a constant size of 3312 bits. From the reported timings, it is evident that we prove the satisfaction of arithmetic circuits at a rate of $\approx$ 77.8 constraints/ms on our considered setup.

Whereas the complexity of VC methods that fully convert programs into circuits to verify each emulated CPU cycle's correctness grows with a program's control-flow and assembly complexity as described in Section 2, our coarse-grained approach grows only to the control-flow complexity. For example, the cost per CFG edge (i.e., control-flow) is ≈ 674 constraints for the fifth circuit in Table 3, which is even less than the per-cycle cost of $\approx 1,458$ in [6]. (Note also that there are inherently more CPU cycles than control-flow transitions during a program's execution.) Thus, while we require a trust anchor on the prover to record and authenticate a program's execution path, we scale better to larger programs. Finally, contrary to circuits crafted for particular programs, our circuit allows attesting to the execution of arbitrary programs via its inputs.

Attesting to executed instructions. While generally not considered for CFA since the attested program is generally considered to reside in DEP-protected memory, [53] proposed having the prover hash the executed instructions along with the BBL addresses to detect TOCTOU attacks when considering physical (non-invasive) adversaries who can manipulate program code during runtime. In this case, the only change in the recorded execution path $\mathcal{EP}$ is that it contains digests instead of the destination and return addresses, i.e., it becomes a sequence of transitions of the form $(jmpkind, d_{dst}, d_{ret})$, where $d = \mathsf{H}(addr||instructions)$. We can easily support this approach by initially storing $\mathsf{H}(addr||BBL_{addr})$ as the elements of the translator $\mathcal{M}$ instead of only storing the addresses. Further, note that only the hashing of the attested execution path $\mathcal{EP}$ and the translator $\mathcal{M}$ are affected (rows 3 and 4 of Table 1). Specifically, assuming 88-bit digests as in [53], each transition will occupy 178 bits (destination and return address and 2 bits for the jumpkind).

Other proof systems. The jsnark interface alternatively supports libsnark's implementation of the optimized Pinocchio zkSNARK proof system [42] as proposed in [6]. However, using this latter system [6] over Groth16 [24] showed an increase of $\approx$ 13% in proving time and $\approx$ 45% in proving key size for the largest circuit in Table 3, including a larger proof size of 2294 bits (7 $\mathbb{G}_1$ elements and 1 $\mathbb{G}_2$ element). While current SNARK technology is on the borderline

Table 3: This table shows the average time (and standard deviation, σ), after 10 iterations, for the worker to generate proofs over ZEKRA circuits compiled to support different attestation data sizes. Proof verification takes ≈ 2 ms in all cases.

Circuit Config (Data Structure Sizes)							Compiled Circuit w. Component Workload Dist. (in %)								Worker
E	N	D	ℓ	*label*	*bucket*	*addr*	pk_C (MB)	# Const.	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	Prove (avg. s)
500	500	15	15	10 bits	7 bits	24 bits	64.638	336,230	7.6	1.6	0.8	38.3	32.1	19.6	4.316 ($\sigma \approx 0.007$)
500	500	15	15	10 bits	7 bits	88 bits	69.327	366,605	7.0	7.0	3.5	35.1	29.5	18.0	4.709 ($\sigma \approx 0.001$)
1000	1000	15	15	10 bits	7 bits	24 bits	134.180	703,669	7.3	1.5	0.7	39.3	32.5	18.7	8.974 ($\sigma \approx 0.057$)
1000	1000	15	15	10 bits	7 bits	88 bits	143.888	764,419	6.7	6.7	3.3	36.1	29.9	17.3	9.730 ($\sigma \approx 0.008$)
1200	1000	15	15	10 bits	7 bits	24 bits	158.907	809,043	6.3	1.6	0.7	39.4	32.5	19.6	10.552 ($\sigma \approx 0.049$)
1200	1000	15	15	10 bits	7 bits	88 bits	170.306	877,893	5.8	7.0	2.9	36.3	30.0	18.0	11.374 ($\sigma \approx 0.026$)

of feasibility, proof systems are evolving increasingly, and thus we expect to handle (and optimize) larger arithmetic circuits more efficiently shortly. Furthermore, we note that the workload requirements of the worker can be further scaled up using current systems like DIZK [52], which allows the generation of proofs to be distributed across machines (workers) in a compute cluster (e.g., EC2). Additionally, note that proof systems have also recently emerged that outperform the Groth16 proof system, such as SpartanSNARK [45], which, compared to libsnark's implementation of Groth16, appears to be 2× faster on the worker. See the corresponding works for details on their proving time and key size complexities.

8 DISCUSSION AND SECURITY PROPERTIES

Comparison with CFA works. While our scheme suffers on the intermediate worker due to the computational resources currently needed to generate zkSNARK proofs[3], we achieve (i) optimal cost on verifiers, (ii) optimal transmission overhead (towards the verifier), and (iii) stronger security properties, as opposed to all existing CFA works [2, 3, 15, 16, 28, 34, 37, 41, 49, 50, 53, 54]. Both (i) and (ii) are due to the "succinctness" of zkSNARK proof constructions, where verification is unaffected by circuit complexity, and the proof size is constant, e.g., with Groth16, the proof only contains three group elements (totaling 1019 bits). Moreover, besides the proof, the verifier only needs to receive the circuit's public inputs, which all have constant sizes, comprising the considered entry and exit node labels, three digests, and the verifier's nonce (whose echo signifies freshness). Note that in other CFA schemes, the prover generally transmits the full execution path and a corresponding digest directly to the verifier. Furthermore, with all other CFA schemes, all verifiers are assumed to maintain an extensive reference database of all the possible execution paths [2] (or, more commonly, the attested program's CFG [16] due to the difficulty of exhaustively discovering all such paths beforehand as described in Section 3.3) to check the legality of attested execution paths. We cover (iii) in Section 8.1.
Execution path compression. While not related to our protocol's effectiveness, the considered granularity of the program CFG and execution paths directly affects the efficiency and scalability of our approach. As also noted by other works [2, 15, 53], we can, without sacrificing accuracy, decrease granularity to increase code coverage by pruning unnecessary edges in a CFG and ignoring repetitions in the execution path. For example, to simplify a CFG, we can, similar to inlining, where callee functions are inlined into the caller functions to reduce complexity, fuse connected nodes where only one path exists between the nodes, e.g., the nodes n_2, n_3 and n_7 in Fig. 1. This, however, requires that the prover's trust anchor can identify and translate execution paths into their succinct form, e.g., if it observes the path $n_1 \rightsquigarrow n_3 \rightsquigarrow n_8 \rightsquigarrow n_4$ during program execution, then it records it as a function call from n_1 to n_8 and a function return to n_4. Similarly, to mitigate path explosion caused by loops, the prover can notice when a sub-path begins consecutively repeating itself, e.g., $n_5 \rightsquigarrow n_6 \rightsquigarrow n_5 \rightsquigarrow n_6$, and record it as only occurring once as considered in our evaluation in Section 7.2.
Limitations. However, on a contrary note, while the current ZEKRA circuit puts no restraints on the number of loop iterations, ensuring a correct (or safe) number of loop iterations can be significant depending on the application [2]. Thus, we note the possibility of extending the circuit to accept a secret policy specifying such loop bounds. Furthermore, while we currently assume a *semi*-untrusted worker, future work can weaken this trust assumption as described in Appendix A. Finally, besides further optimizations to reduce the computational resources needed by workers, future work should investigate the viability of constructing a similar scheme with post-quantum-secure proof systems, e.g., using STARKs [7].

[3]Which makes it challenging to target complex software, such as that considered by ScaRR [50] and ReCFA [54], using our scheme due to the worker resources needed.

8.1 Security Properties

Besides its secure implementation, our scheme's foundational security is given by the underlying proof system's security (and circuit compiler). Note that for all configurations we ran our prototype on (hundreds of executions), we recorded no completeness errors.
Unforgeability. Besides negligible error probability for falsely rejecting or falsely accepting an execution path or claims about an execution path as authentic, our scheme is secure against (computationally bounded) adversaries attempting to forge execution paths due to the utilization of an accompanying signature scheme with the zkSNARK proof system. For example, suppose a dishonest worker (i) uses the prover's signed digest as a public input during proof generation but uses a different execution path, nonce, or random noise, as the secret input, or (ii) additionally computes a digest over these values to use as the public input instead of the prover's digest. In the first case (i), the verifier rejects the proof in the proof verification stage due to unmet constraints, and in the latter case (ii), the verifier rejects the proof in the signature verification stage due to a digest mismatch. Therefore, a dishonest worker cannot convince the verifier that an illegal (forged) execution path (e.g., one that reveals an attack against the attested program) is legal.
Proofs are zero-knowledge. Our scheme ensures that a verifier only learns whether the expected prover freshly recorded an execution path and whether the execution path is legal according

to the secret $\mathcal{AL}$ preimage of a specific trusted reference digest. Specifically, we ensure that no execution path or CFG details are disclosed to the verifier by accepting them as secret circuit inputs and making the reference digests statistically-hiding commitments.

Forward & back edge integrity. The ZEKRA circuit is only satisfied if the provided execution path is legal according to the given $\mathcal{AL}$. The rules are simple. Transitions target adjacent nodes, and back transitions are shadow stack compliant. By verifying that the execution path consistently flows through adjacent nodes in the forward direction (using the adjacency list) and that callees always return to the contextually correct caller (using the shadow stack), we ensure fine-grained detection of all control-based attacks. (see Appendix E for a more extensive description of how the different control-flow-based attacks result in rejection on the verifier).

Execution path completeness. Like how a CFG precisely models all program executions, we support attesting to any execution path.

9 CONCLUSIONS

We presented ZEKRA, a novel and effective protocol that utilizes the combined strength of verifiable computation and control-flow attestation to enable underpowered provers to convince untrusted verifiers about the correct control-flow execution of deeply embedded programs in zero knowledge. The proposed scheme guarantees the attested program's forward and back-edge correctness according to its reference CFG, using several optimizations for representing and traversing CFGs. While currently only demonstrated for deeply embedded applications, our research suggests verifiable computation based on zkSNARK constructions as a feasible direction for enhancing CFA schemes with additional privacy guarantees.

ACKNOWLEDGMENTS

The European Commission supported this work under the CONNECT and ASSURED projects; Grant Agreements 101069688 and 952697.

REFERENCES

[1] Martín Abadi et al. 2009. Control-flow integrity principles, implementations, and applications. *ACM TISSEC* 13, 1 (2009), 1–40.
[2] Tigist Abera et al. 2016. C-FLAT: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference.* 743–754.
[3] Tigist Abera et al. 2019. DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems. In *NDSS.*
[4] Alcardo Alex Barakabitze, Arslan Ahmad, Rashid Mijumbi, and Andrew Hines. 2020. 5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges. *Computer Networks* 167 (2020).
[5] Eli Ben-Sasson et al. 2013. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Annual cryptology conference.* Springer, 90–108.
[6] Eli Ben-Sasson et al. 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In *23rd {USENIX} Security Symposium.* 781–796.
[7] Eli Ben-Sasson et al. 2018. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive* (2018).
[8] Daniel Benarroch et al. 2021. Zero-Knowledge Proofs for Set Membership: Efficient, Succinct, Modular. In *International Conference on Financial Cryptography and Data Security.* Springer, 393–414.
[9] Tyler Bletsch et al. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM AsiaCCS.* 30–40.
[10] Sean Bowe, Ariel Gabizon, and Matthew D Green. 2018. A multi-party protocol for constructing the public parameters of the Pinocchio zk-SNARK. In *International Conference on Financial Cryptography and Data Security.* Springer, 64–77.
[11] Benjamin Braun et al. 2013. Verifying computations with state. In *Proceedings of the twenty-fourth ACM Symposium on Operating Systems Principles.* 341–357.
[12] Xavier Carpent et al. 2018. Reconciling Remote Attestation and Safety-Critical Operation on Simple IoT Devices. In *DAC '18.*
[13] Liqun Chen et al. 2008. Property-based attestation without a trusted third party. In *International Conference on Information Security.* Springer, 31–46.
[14] Heini Debes. 2022. Code for ZEKRA. https://github.com/HeiniDebes/ZEKRA
[15] Ghada Dessouky et al. 2017. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Design Automation Conference 2017.* 1–6.
[16] Ghada Dessouky et al. 2018. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM ICCAD.* IEEE, 1–8.
[17] Jacob Eberhardt and Stefan Tai. 2018. Zokrates-scalable privacy-preserving off-chain computations. In *IEEE International Conference on Internet of Things.*
[18] Embench. 2022. Modern Embedded Benchmark Suite. https://www.embench.org/
[19] Aurélien et. al. Francillon. 2014. A minimalist approach to Remote Attestation. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).*
[20] Rosario Gennaro et al. 2013. Quadratic span programs and succinct NIZKs without PCPs. In *EUROCRYPT.* Springer, 626–645.
[21] Shafi Goldwasser et al. 1989. The knowledge complexity of interactive proof systems. *SIAM Journal on computing* 18, 1 (1989), 186–208.
[22] Lorenzo Grassi et al. 2021. Poseidon: A new hash function for zero-knowledge proof systems. In *30th {USENIX} Security Symposium ({USENIX} Security 21).*
[23] Jens Groth. 2010. Short pairing-based non-interactive zero-knowledge arguments. In *Asiacrypt.* Springer, 321–340.
[24] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *EUROCRYPT.* Springer, 305–326.
[25] Aric Hagberg et al. 2008. *Exploring network structure, dynamics, and function using NetworkX.* Technical Report. Los Alamos National Lab., Los Alamos, NM.
[26] Caleb Helbling. 2020. Directed Graph Hashing. *preprint arXiv:2002.06653* (2020).
[27] Hong Hu et al. 2016. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *IEEE Symposium on Security and Privacy (SP).*
[28] Jianxing Hu et al. 2019. A probability prediction based mutable control-flow attestation scheme on embedded platforms. In *18th TrustCom/BigDataSE.* IEEE.
[29] iden3. 2022. zkSNARK implementation. https://github.com/iden3/snarkjs
[30] Joe Kilian. 1992. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing.*
[31] Tommy Koens, Coen Ramaekers, and Cees Van Wijk. 2018. Efficient zero-knowledge range proofs in ethereum. *ING. blockchain@ ing. com* (2018).
[32] Ahmed Kosba. 2021. Java zkSNARK library. https://github.com/akosba/jsnark
[33] Ahmed Kosba et al. 2018. xJsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP).* IEEE, 944–961.
[34] Boyu Kuang et al. 2020. DO-RA: data-oriented runtime attestation for IoT devices. *Computers & Security* 97 (2020), 101945.
[35] SCIPR Lab. 2020. C++ zkSNARK library. https://github.com/scipr-lab/libsnark
[36] Panagiotis Liakos et al. 2017. Realizing memory-optimized distributed graph processing. *IEEE Transactions on Knowledge and Data Engineering* 30, 4 (2017).
[37] Jingbin Liu et al. 2019. Log-Based Control Flow Attestation for Embedded Devices. In *International Symposium on Cyberspace Safety and Security.* Springer, 117–132.
[38] Silvio Micali. 1994. CS proofs. In *Proceedings 35th Annual Symposium on Foundations of Computer Science.* IEEE, 436–453.
[39] Assa Naveh and Eran Tromer. 2016. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *2016 IEEE S&P.* IEEE, 255–271.
[40] Ivan De Oliveira Nunes et al. 2020. {APEX}: A Verified Architecture for Proofs of Execution on Remote Devices under Full Software Compromise. In *29th {USENIX} Security Symposium ({USENIX} Security 20).* 771–788.
[41] Ivan De Oliveira Nunes et al. 2020. Tiny-CFA: A minimalistic approach for CFA using verified proofs of execution. *arXiv:2011.07400* (2020).
[42] Bryan Parno et al. 2013. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 238–252.
[43] Eli Ben Sasson et al. 2014. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy.* IEEE, 459–474.
[44] Berry Schoenmakers et al. 2016. Trinocchio: privacy-preserving outsourcing by distributed verifiable computation. In *ACNS.* Springer, 346–366.
[45] Srinath Setty. 2020. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Annual International Cryptology Conference.* Springer, 704–737.
[46] Srinath Setty et al. 2012. Taking proof-based verified computation a few steps closer to practicality. In *21st {USENIX} Security Symposium.* 253–268.
[47] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *14th ACM CCS.*
[48] Yan Shoshitaishvili et al. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. (2016).
[49] Zhichuang Sun et al. 2020. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 1433–1449.
[50] Flavio Toffalini et al. 2019. ScaRR: Scalable Runtime Remote Attestation for Complex Systems. In *22nd International Symposium on {RAID}.* 121–134.
[51] Riad S Wahby et al. 2014. Efficient RAM and control flow in verifiable outsourced computation. *Cryptology ePrint Archive* (2014).
[52] Howard Wu et al. 2018. {DIZK}: A distributed zero knowledge proof system. In *27th {USENIX} Security Symposium ({USENIX} Security 18).* 675–692.
[53] Shaza Zeitouni et al. 2017. Atrium: Runtime attestation resilient under memory attacks. In *2017 IEEE/ACM ICCAD.* IEEE, 384–391.
[54] Yumei Zhang et al. 2021. ReCFA: Resilient Control-Flow Attestation. In *Annual Computer Security Applications Conference (ACSAC '21).*

A TRUST ASSUMPTIONS

On the Trustworthiness of the Worker. Our scheme considers a resource-constrained prover device capable of securely recording and signing the execution path taken by a program during runtime. While the prover wants to assure a verifier that the program was executed correctly (i.e., in the absence of control-flow attacks), the prover wants to keep the path and program details *private*. Therefore, the prover outsources the signed execution path to a resourceful worker who produces a zero-knowledge proof of the path's correctness which is publicly verifiable and does not reveal the execution path. However, since the prover gives the secret inputs (i.e., the recorded execution path and blinding factors) to the worker (who must also know all reference materials) for producing the proof, it must trust that the worker keeps the inputs secret. In other words, we trust the worker regarding the posterior privacy of the secret circuit inputs to satisfy our protocol's privacy goals as described in Section 4. However, the worker is *not a trusted party* since the worker can attempt to cheat in producing the proof (e.g., by tampering with the inputs), which is why we require verifiable computation to let verifiers detect such *dishonest* behavior. Therefore, since the worker is trusted regarding posterior input privacy but untrusted regarding proof generation, we refer to the worker as semi-dishonest. For future work, we note the possibility of further weakening our trust assumptions (on the worker) by hiding the secret inputs, e.g., by employing multiparty computation techniques based on Shamir's secret sharing [44].

Minimal Trusted Computing Base. As mentioned in Section 4, a prover is a low-end embedded device with limited resource capabilities. Hence, it is highly desirable to restrict the influence of the underlying trusted component (i.e., trust anchor) on the normal execution of the host, which is a common barrier affecting the generality and applicability of existing remote attestation schemes to safety-critical systems [12]. Thus, ZEKRA's design choice is to follow a minimalist attestation approach [19], assuming the existence of a root-of-trust with only those properties needed to attain remote attestation services. These include recording a program's execution path and cryptographic functions for signing the recorded execution path to guarantee origin authentication. Note that for the latter, the resource overhead is mainly determined by the hashing since the signing operation is independent of the execution path size, as only a fixed-size hash is signed. For the former, one natural way to extract an executing program's execution path is to equip devices with tracing capabilities, e.g., by leveraging existing processor hardware features and commonly-used IP blocks, as done by the tracer proposed in LO-FAT [15]. Note that while tracing program execution is relatively efficient with minimal perturbation, interpreting (i.e., translating in our case) raw memory addresses and verifying the recorded execution path's correctness is complex as it relies on additional trusted reference materials. ZEKRA decouples these two functionalities (recording and translating/verification). Thus, ZEKRA enables a minimal trust anchor that only needs recording support without additional decoding capabilities. Specifically, the trust anchor only records raw traces (capturing sequences of memory addresses visited), which are then sent to the worker for the more complex and program-specific task of convincing the verifier that the execution path was correct according to a specific program's trusted reference materials. Resolving this inherent link between tracing and program-specific decoding, ZEKRA supports devices with continuous (non-intrusive) tracing capabilities (e.g., ARM Coresight) which offer negligible impact on the performance of the programs executing in the normal world. Specifically, note that recording execution paths is an efficient and program-agnostic process and, thus, can easily fit inside a small trust anchor. Especially since only the hash of the recorded execution path, which is being accumulated during program execution, needs to be securely stored, while the path itself can reside in unprotected memory, as done in most CFA schemes. However, since the verification is program-dependent, it would require the trust anchor to maintain all trusted reference materials (i.e., CFGs and translator mappings, which can grow large for complex binaries, and also all possible entry/exit node pairs, including semantical information to determine which pair to consider for each attestation) in *secure memory* for each of the programs (and attestable segments thereof) offered by a prover. We would also require additional mechanisms on provers to guarantee attestation freshness to verifiers. Furthermore, the trust anchor's responsibility becomes even more complex if we consider systems with the possibility of over-the-air updates since the trust anchor would need additional logic to update its trusted reference materials. Therefore, it becomes clear that reconciling the needs of safety-critical applications and remote attestation security requirements through a minimal architecture for the underlying trust anchor enables ZEKRA to support practical remote attestation with minimal requirements over the prover's computational resources. In this process, we remain agnostic regarding the underlying hardware by making the fewest possible assumptions about specific devices, thus pushing towards a more lightweight blueprint that can be realized on a wide range of low-end devices, with minimal modifications and assumptions on required secure hardware.

B HIGH-LEVEL PROGRAMS AS CIRCUITS

To illustrate what we mean by cost, first imagine the ZEKRA algorithm in Fig. 3 as a standard program that we want to transform into an arithmetic circuit. When the code is fed to a circuit compiler, it will flatten out the code by unrolling the loop to the worst-case number of iterations (as determined by the size of the execution path data structure, set to some value E) while taking each branch of each conditional statement into account for each iteration. Then, the compiler will convert the code to single static assignments, which are then transformed into one or more constraints [11, 51]. The result is a concise set of constraints (or equations) that is satisfied only when all variables hold all equations simultaneously. For a simple statement "$a = b$", the equivalent constraint set might consist of the constraints representing a, the constraints representing b, and an additional constraint relating the outputs of a and b [46].

C EFFICIENT GRAPH TRAVERSAL IN A LINEAR SYSTEM OF EQUATIONS

While we can use $\mathcal{AL}'$ to efficiently (i.e., inexpensively) access n_{cur}'s encoded neighbors as we iterate over the execution path as described in Section 6.1, which operations we use to check whether a particular neighbor exists directly impacts the circuit's performance. For example, using the mathematical equations outlined in

Section 5.1, we can effectively determine if n_{dst} is listed as n_{cur}'s neighbor at the execution path's ith transition by checking that $\mathcal{AL}'(i)(l) = \lfloor n_{dst}/8 \rfloor$ and $\mathcal{AL}'(i)(l+1)[n_{dst} \bmod 8] = 1$ hold for some level j, where j is an even number and $0 \leq j < 2\ell - 1$. However, while the integer division and modulo operations, $\lfloor n_{dst}/8 \rfloor$ and $n_{dst} \bmod 8$, are not exceptionally expensive due to the constant divisor (arbitrary modulo operations, however, are expensive, especially on prime fields as they require a range check: $a = qb + r, r < b$), the latter equation assumes bitwise operations to access bit ($n_{dst} \bmod 8$) of *rems*, which can become prohibitively expensive if done repeatedly. Specifically, to access individual bits of an n-bit integer-carrying wire, e.g., *rems*, in the circuit, we require, at a minimum, a similar number of constraints to cover each bit of the wire. Thus, to alleviate the need to access individual bits using bitwise operations, we transform the equations into pure algebraic expressions as shown in (3), where the left-hand side of the latter equation mathematically accesses the value of the ($n_{dst} \bmod 8$)th bit of *rems* by first pruning the bottom ($n_{dst} \bmod 8$) bits using integer division and then cherry-picking the low order bit of the result using modulo.

$$\lfloor n_{dst}/8 \rfloor = bucket$$

$$\lfloor rems/2^{n_{dst} \bmod 8} \rfloor \bmod 2 = \begin{cases} 0, & \text{bit is unset} \\ 1, & \text{bit is set} \end{cases} \tag{3}$$

Thus, if we find a pair $(bucket, rems) \in \mathcal{AL}'(i)(n_{cur})$ satisfying (3) for some transition i's destination node n_{dst}, we know that n_{dst} is a valid neighbor of n_{cur}, i.e., $n_{cur} \rightsquigarrow n_{dst}$ is legal. The remaining challenge, however, is that the components of the latter equation in (3) require expensive *variable* integer division and *variable* exponentiation.

However, applying the power of SNARK verification, rather than doing the variable integer division $\lfloor rems/2^{n_{dst} \bmod 8} \rfloor$ and variable exponentiation $e = 2^{n_{dst} \bmod 8}$ in the circuit, we make the prover supply the answer of the integer division (the quotient, q_1, and remainder r_1) and exponentiation e, and then in the circuit we verify that $eq_1 + r_1 = rems$, where $r_1 < e$, and e is the expected product of $2^{n_{dst} \bmod 8}$. Note that we can efficiently check whether e is the expected product since we work with a constant divisor of 8 and can therefore create a fixed sequence of all possible products: $P = (2^0, \dots, 2^7)$. Thus, in the circuit we only need to check that $\exists i : e = P_i \wedge i = n_{dst} \bmod 8$ holds, which is integrated as a linear search over the elements of P and made efficient by favoring the discounted price of math operations in the native field $\mathbb{F}$ over a conditional statement, i.e., we program the lookup as: $\prod_{i=0}^{7}((n_{dst} \bmod 8) - i) + (e - P_i)$, which becomes zero if e is as expected for n_{dst}. Further, to account for the modulo 2 in (3), we make the prover supply the answer to the integer division $q_2 = \lfloor q_1/2 \rfloor$ and corresponding remainder $r_2 = q_1 \bmod 2$, which we verify inside the circuit by checking that $2q_2 + r_2 = q_1$. Note here that r_2 corresponds to the left-hand side computation of the latter equation in (3). Thus, we know to accept n_{dst} as a neighbor if all equations hold and $r_2 = 1$.

In total, the circuit accepts five additional secret inputs, e, q_1, r_1, q_2, r_2, which work as hints for the circuit to more efficiently check whether a particular destination node n_{dst} is legal by verifying a system of linear equations. Similar to the adjacency list $\mathcal{AL}'$, which contains n_{cur}'s $(bucket, rems)$ pairs pre-split for every transition as we walk over the execution path, the circuit accepts the additional five secret inputs that prove a transition's destination node's validity as a two-dimensional (sequentially accessed) array, π_{exists}, which is also ordered by transitions. See Table 4 for a summary of the computation and verification of the input variables/hints.

Table 4: Auxiliary variables (hints) that allow the circuit to efficiently verify that a transition's destination node n_{dst} exists in the current node n_{cur}'s encoded neighbor list. Basically, for some $(bucket, rems) \in \mathcal{AL}'(n_{cur})$:

Worker computes	Variable in π_{exists}	Circuit verifies
$2^{n_{dst} \bmod 8}$	e	$e = 2^{n_{dst} \bmod 8}$
$\lfloor rems/e \rfloor$	q_1	$eq_1 + r_1 = rems$
$rems \bmod e$	r_1	$r_1 < e$
$\lfloor q_1/2 \rfloor$	q_2	$2q_2 + r_2 = q_1$
$q_1 \bmod 2$	r_2	$r_2 = 1$
		$bucket = n_{dst}/8$

D FINAL DESIGN OF THE ZEKRA CIRCUIT

Fig. 4 shows the high-level code with the optimizations described in Section 6.1 and Appendix C, which is compiled into the ZEKRA circuit. As secret input, the circuit accepts: an encoded adjacency list $\mathcal{AL}$ of length N (or padded to equal N) representing the attested program's CFG, some execution path $\mathcal{EP}$ of length E (or padded to equal E) corresponding to the recorded execution path, the mapping $\mathcal{M}$ for translating the attested program's addresses into numeric labels, the random padding used as a blinding factor for the adjacency list (r_1), execution path (r_2), and mapping (r_3), respectively, the auxiliary adjacency list $\mathcal{AL}'$ containing the decoded $bucket, rems$ pairs of the expected $\mathcal{AL}$ entry for each transition, the translation of the execution path addresses into numeric labels $\mathcal{L}$, and finally, the auxiliary proofs π_{exists} to help verify each transition's destination node's validity. As public input, the circuit accepts: the digests of the adjacency list (h_1), execution path (h_2), and the mapping (h_3), respectively, an initial node $n_{\triangleright}$, a final node $n_{\blacktriangleleft}$, and the verifier's nonce *nce*. Note that to transform our high-level program into a circuit, we must clearly define the bounds of all data structures we want to be expressed (i.e., for which we want constraints to be generated). However, while we can easily pad adjacency lists to match N before supplying them as input (thus supporting different program complexities with the same circuit), the same does not immediately apply to the execution paths. Note that the execution path through a program might vary drastically between different executions. Thus, to support varying sizes of execution paths, it must be padded to the appropriate length (E) by appending "empty" transitions, which is detected in the circuit when $jmpkind = 2$. Furthermore, to allow translating padded transitions using $\mathcal{M}$, we reserve an entry ($\mathcal{M}+1$) whose value is used as both the destination and return address in each padded transition.

Given the inputs, we verify that the secret adjacency list, attested execution path, and mapping are indeed the correct preimages of the corresponding public digests by first compressing and then hashing each data structure using our implementation of the circuit-friendly

```
// public and secret circuit inputs
public{$h_1, h_2, h_3, n_{\triangleright}, n_{\blacktriangleleft}, nce$}
secret{$\mathcal{AL}[N], \mathcal{EP}[E][3], \mathcal{M}[N+1], r_1, r_2, r_3,$
        $\mathcal{AL}'[E][2\ell], \mathcal{L}[E][2], \pi_{exists}[E]$} // hints
external{ // code executed by worker to compute hints
    $n_{cur} \leftarrow n_{\triangleright}$ // keep state during traversal
    for $i = 0 \ldots E$ do // compute hints for each step
        $\mathcal{AL}'(i) \leftarrow$ split$(\mathcal{AL}(n_{cur}))$
        $\mathcal{L}(i)(dst) \leftarrow \{j | \mathcal{M}(j) = \mathcal{EP}(i)(dst)\}$
        $\mathcal{L}(i)(ret) \leftarrow \{j | \mathcal{M}(j) = \mathcal{EP}(i)(ret)\}$
        $\pi_{exists} \leftarrow$ Table 4$(\mathcal{EP}(i)(dst), \mathcal{AL}'(n_{cur}))$
        $n_{cur} \leftarrow \mathcal{L}(i)(dst)\}$
// circuit code
$shadowStack[D]$
$\mathsf{Vf}(\mathsf{H}(\mathcal{AL} \| r_1) = h_1)$
$\mathsf{Vf}(\mathsf{H}(\mathcal{EP} \| nce \| r_2) = h_2)$
$\mathsf{Vf}(\mathsf{H}(\mathcal{M} \| r_3) = h_3)$
$\mathsf{Vf}(\forall i \in \{0 \ldots E\} : \mathcal{M}(\mathcal{L}(i)(dst)) = \mathcal{EP}(i)(dst) \wedge$
                            $\mathcal{M}(\mathcal{L}(i)(ret)) = \mathcal{EP}(i)(ret))$
$n_{cur} \leftarrow n_{\triangleright}$ // keep state during traversal
for $i = 0 \ldots E$ do // walk the execution path
    $jmpkind \leftarrow \mathcal{EP}(i)(jmpkind)$
    $(n_{dst}, n_{ret}) \leftarrow (\mathcal{L}(i)(dst), \mathcal{L}(i)(ret))$
    if $jmpkind \neq \emptyset$ then // not an empty (padded) transition
        $bucket \leftarrow \lfloor n_{dst}/8 \rfloor$
        $pos \leftarrow n_{dst} \bmod 8$
        $e, q_1, r_1, q_2, r_2 \leftarrow \pi_{exists}(i)$
        $\mathsf{Vf}\Big(\Big(\sum_{j=0,2,\ldots}^{2\ell-1} \mathcal{AL}'(i)(j+1)2^{\lfloor j/2 \rfloor(bucket\ \text{bitwidth}+8)+8}$
            $+ \mathcal{AL}'(i)(j)2^{\lfloor j/2 \rfloor(bucket\ \text{bitwidth}+8)}\Big) = \mathcal{AL}(n_{cur})\Big)$
        $\mathsf{Vf}\left(\left(\prod_{j=0}^{7}(pos - j) + (e - P_j)\right) = 0\right)$
        $\mathsf{Vf}(2q_2 + r_2 = q_1)$
        $\mathsf{Vf}(r_2 = 1)$
        $\mathsf{Vf}(\exists j \in \{0, 2, \ldots, 2\ell - 2\} :$
            $\mathcal{AL}'(i)(j) = bucket \wedge$
            $\mathcal{AL}'(i)(j+1) = eq_1 + r_1)$
        $n_{cur} \leftarrow n_{dst}$ // progress CFG state
    if $jmpkind = call$ then
        push$(n_{ret}, shadowStack)$
    elseif $jmpkind = ret$ then
        $\mathsf{Vf}($pop$(shadowStack) = n_{dst})$
$\mathsf{Vf}(n_{cur} = n_{\blacktriangleleft})$
```

Figure 4: The high-level ZEKRA program code, which can be compiled into an outsourceable circuit.

Poseidon [22] hashing function H as described in Section 5.1. Assuming that the digests were correct, the circuit proceeds to verify that the worker's translation $\mathcal{L}$ of the execution path addresses into their corresponding numeric labels was done correctly according to $\mathcal{M}$. Then, knowing that $\mathcal{L}$ accurately reflects the attested execution path in the abstract world of $\mathcal{AL}$, the circuit proceeds to traverse $\mathcal{AL}$ using $\mathcal{L}$ to verify that the execution path: (i) began at the expected $n_{\triangleright}$, (ii) contains only transitions that are conformant to the adjacency list, and (iii) ends at the expected final node $n_{\blacktriangleleft}$. The traversal is done by instantiating a state variable n_{cur} to $n_{\triangleright}$, which, as we iterate over each transition $(jmpkind, n_{dst}, n_{ret})$ in the execution path, we verify that n_{dst} is a valid neighbor of n_{cur} by consulting n_{cur}'s auxiliary adjacency list, as follows. We first retrieve n_{cur}'s encoded neighbors by accessing n_{cur}'s encoded entry in $\mathcal{AL}$ (which is expensive since RAM is expensive), which we compare against the current transition's n_{dst}'s entry in $\mathcal{AL}'$ (whose access is cheap since we access it sequentially). If the entries match, we know that we can securely rely on $\mathcal{AL}'$ to accurately report n_{cur}'s neighbors, which we leverage to efficiently verify whether n_{dst} is a valid neighbor of n_{cur} by verifying that the current transition's proof in π_{exists} is valid with respect to some *bucket*, *rems* pair in $\mathcal{AL}'$ as described in Section C. If n_{dst} is determined to be a valid neighbor, we update n_{cur} to n_{dst} and progress to the next transition. Moreover, while iterating over the path, we maintain a shadow stack to ensure exact back edge integrity as described in Section 6. Finally, we conclude by verifying that n_{cur} reached $n_{\blacktriangleleft}$. The circuit is only satisfied if *all* verifications were successful.

E REJECTION OF CONTROL-FLOW ATTACKS

To evaluate ZEKRA in terms of detecting control-flow attacks (i.e., preventing proofs from being accepted when execution paths are illegal), we tested several paths bearing real code injection or ROP/JOP attack patterns. Note that the ZEKRA circuit, when compiled, is a concise set of constraints (or equations) that is satisfied only when *all variables hold all equations* simultaneously. This constraint system includes both the set of constraints from the forward-edge integrity component, which requires jumps and calls to target *valid neighbors*, and the set of constraints from the backward-edge component requiring returns to target the *contextually-valid* node. Thus, if an execution path contains *any* transition that causes *any* constraint to fail, then the verifier rejects the proof. Thus, as expected, conventional control-flow hijacking attacks that employ code injection result in rejected proofs since they add transitions to the execution path which target nonexistent CFG nodes.

Similarly, code-reuse attacks such as ROP and JOP are detected and rejected by the verifier since they cause an execution path to contain transitions that target invalid neighbors. Furthermore, if a function's return address is hijacked to execute a malicious gadget sequence, then this will also cause an unfulfilled constraint in the backward-edge integrity component since the shadow stack's topmost entry will not match this new node. Furthermore, note that the execution path must also, at a minimum, be translatable to its numeric representation according to the program-specific address-to-label mapper $\mathcal{M}$. Therefore, in the case of control-flow hijacking, e.g., when stitching together a chain of gadgets for ROP, where an adversary might include branches to unexpected offsets within a BBL instead of its starting address, this is always caught since $\mathcal{M}$ will not include such entries. However, note that the circuit is limited to detecting control-flow attacks, i.e., DOP attacks (see Fig. 1), even impure, will remain undetected unless the reference CFG forces legal executions through designated routes in the CFG.