Zero Knowledge Static Program Analysis

Zhiyong Fang
Texas A&M University
zhiyong.fang.1997@tamu.edu

David Darais[*]
Galois, Inc.
darais@galois.com

Joseph P. Near
University of Vermont
jnear@uvm.edu

Yupeng Zhang
Texas A&M University
zhangyp@tamu.edu

ABSTRACT

Static program analysis tools can automatically prove many useful properties of programs. However, using static analysis to prove to a third party that a program satisfies a property requires revealing the program's source code. We introduce the concept of *zero-knowledge static analysis*, in which the prover constructs a zero-knowledge proof about the outcome of the static analysis *without revealing the program*. We present novel zero-knowledge proof schemes for intra- and inter-procedural abstract interpretation. Our schemes are significantly more efficient than the naive translation of the corresponding static analysis algorithms using existing schemes. We evaluate our approach empirically on real and synthetic programs; with a pairing-based zero knowledge proof scheme as the backend, we are able to prove the control flow analysis on a 2,000-line program in 1,738s. The proof is only 128 bytes and the verification time is 1.4ms. With a transparent zero knowledge proof scheme based on discrete-log, we generate the proof for the tainting analysis on a 12,800-line program in 406 seconds, the proof size is 282 kilobytes, and the verification time is 66 seconds.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**.

KEYWORDS

Zero knowledge proofs; Static program analysis

ACM Reference Format:
Zhiyong Fang, David Darais, Joseph P. Near, and Yupeng Zhang. 2021. Zero Knowledge Static Program Analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3460120.3484795

[*]Work done in part as a consultant for Stealth Software, Inc.

1 INTRODUCTION

Static program analysis tools are designed to automatically prove properties of programs. Abstract interpretation [28–33] is a widely-used framework for static program analysis; it has been used to verify complex properties in real programs [26, 34, 36, 40, 45, 55, 59, 63–65]. For example, Astrée [34] has been used to verify critical software for aircraft and spacecraft.

Static analysis tools operate on a program's source code, so proving to a third party that a program has a specific property via static analysis requires revealing its source code to the third party. This requirement limits the application of static analysis tools in settings that involve proprietary or otherwise secret algorithms. For example, a credit-scoring company might develop a proprietary scoring algorithm, and want to demonstrate that the algorithm does not discriminate on the basis of race, gender, or other attributes protected by law.[1] A *taint analysis*, easily implemented via abstract interpretation, might be used to demonstrate that the algorithm ignores these factors when generating a score—but the results cannot be checked by a third party *unless the source code is made public.*

This paper introduces *zero-knowledge static analysis*, an approach that allows an untrusted party to prove that a program has a property *without revealing the program.* Recent privacy regulations include a number of important requirements on algorithms for processing data; with zero-knowledge static analysis, organizations will be able to prove compliance of their algorithms *without revealing the algorithms.* For example, a *taint analysis* can be used to prove that consent is obtained before processing data or pseudonyms are used in place of identifying information (to comply with GDPR [5] and COPPA [3]), that Personal Health Information is redacted (to comply with HIPAA [6]), and that exercise of privacy rights does not result in discrimination (to comply with CCPA [2]). A *control-flow analysis* can be used to prove that the program produces proper audit logs and that pseudonymization is performed correctly (to comply with GDPR). Each of these examples represents a large class of properties that are provable with zero-knowledge static analysis for *all possible executions* of the secret program. In addition to demonstrating compliance to regulators, organizations can use the proofs to build trust with the public.

With zero-knowledge static analysis, we envision the applications in practice with the following three steps:

(1) The owner of the program commits to a program P and posts the commitment to the public.

[1]In the US, the Equal Credit Opportunity Act (15 U.S.C. §§1691-1691f)

(2) The owner can then prove some properties of P via our zero-knowledge static analysis scheme (e.g. it does not discriminate) without revealing the source code of P.
(3) Finally, using schemes of zero-knowledge program execution [1, 16, 22], the owner can further proves that executing the same program P on some (public or previously committed) input produces a particular output.

In addition to the discrimination example, other useful applications of zero knowledge static analysis include proving the worst case execution bounds of a program, absence of runtime exceptions, absence of security vulnerabilities related to unsafe memory access (e.g., Heartbleed), or absence of security vulnerabilities due to timing channels (e.g., Spectre [51] and Meltdown [52]).

Zero-knowledge static analysis can also be used to prove the *possible presence of a bug* in a program. There are two ways to demonstrate that a program is likely to have a bug in zero-knowledge: (1) provide a secret input that leads to unexpected behaviors of a program [43], and (2) show that a reasonably precise static analysis for proving the absence of bugs fails. In the first approach, a counterexample can guarantee the presence of a bug, but it is not always easy or possible to find such an input. For example, if a program does not comply with differential privacy [35], one cannot find a counterexample to prove it. Instead, our technique can be used in this case to demonstrate that the program is very likely to not have the property of interest through static analysis, where a more precise analysis makes for stronger evidence than an imprecise one.

Static analysis algorithms are not easily translated into ZKP systems. Recent work in ZKP has resulted in many efficient systems capable of proving arbitrary functions modeled as arithmetic or boolean circuits. Unfortunately, static analysis is typically carried out using stateful algorithms, which are not easily translated directly into circuits. Alternatively, we could translate static analysis algorithms into circuits using existing RAM-based zero-knowledge proof schemes [14, 16, 21, 22, 66, 73] that model the public function as a program in the random-access-memory (RAM) model. However, they usually introduce a high overhead for each instruction in the program (see related work for more details).

1.1 Our Contributions

In this paper, we initiate the study of zero-knowledge static analysis and propose several efficient schemes based on abstract interpretation and worklist algorithms. In particular, our contributions are:

- **Zero-knowledge intra-procedure analysis.** First, we introduce a simple imperative programming language with assignments, branches, loops and memory operations. We propose an efficient zero-knowledge proof scheme for the classic worklist algorithm for abstract interpretations on programs written in our programming language. The prover time of our scheme is $O(T \cdot v + T \log T)$ where v is the number of variables in the program and T is the number of iterations in the worklist algorithm. This is asymptotically optimal up to a logarithmic factor compared to the plain worklist algorithm without zero-knowledge. We apply several important techniques in the literature of RAM-based ZKP such as memory checking and set relationships to construct building blocks of our scheme efficiently.
- **Zero-knowledge inter-procedure analysis.** Second, we extend our scheme to support inter-procedure analysis with function calls. To address new challenges of loops with dynamic sizes, we efficiently realize linked-list operations as a circuit, and apply techniques of loop merge from existing work [66]. The prover time of our zero-knowledge inter-procedure analysis remains $O(T \cdot v + T \log T)$, where v now denotes the maximum number of variables in a function.
- **Implementation and evaluations.** Finally, we implement our zero-knowledge abstract interpretation schemes and evaluate their performance on both real and synthetic programs. As shown in the experiments, we are able to prove the result of the tainting analysis on a program with 12,800 lines of code. It only takes 406s to generate the proof. The proof size is 282KB and the verifier time is 66s. With a different backend in [42], we are able to prove the control flow analysis on a 2,000-line program in 1,738 s, where the proof is only 128 bytes and the verification time is only 1.4ms

1.2 Related Work

Static analysis and abstract interpretation. Static program analysis tools are typically classified as *sound* or *unsound*. Sound tools are capable of *proving* that a program does not contain bugs; unsound tools typically attempt to find as many bugs as possible, but provide no guarantees on the absolute presence or absence of bugs (i.e., there may be both false positives and false negatives). Unsound tools include syntactic bug finders like LINT and FINDBUGS.

The class of sound static analyses includes data flow analyses, (sound) symbolic execution techniques, abstract interpretation, theorem proving, and others. Data flow analyses [46–48] are often used in compilers, as part of optimizations to produce more efficient code. Symbolic execution [50] executes the target program with symbolic input (rather than concrete values), and solves constraints over those symbols whenever conditionals are encountered. Symbolic execution is typically used to determine whether or not it is possible to reach a particular state in the program. Theorem proving techniques often involve manual proof development by the programmer, aided by an automated proof assistant [38, 44].

In this work, we focus on a static analysis approach called *abstract interpretation* [28–33], which is widely used as a foundational framework for both designing program analysis algorithms and proving their soundness. Abstract interpretation-based tools have been used successfully to verify the absence of runtime exceptions in C, C++ and Java programs [34, 59, 65], verify the absence of buffer overruns in C programs [45], verify tight bounds for worst-case execution in real-time systems [36], verify the absence of floating point rounding errors in C and assembly programs [40], verify termination and liveness properties of C programs [26], and compute control flow analysis of higher order functional programs [55, 63]—among hundreds of other applications. In particular, the Astrée tool [34] is well known for its industrial applications, such as its use in verifying the flight control software of the Airbus A340 and A380 fly-by-wire systems, and the automatic docking software of the Jules Vernes ATV, the first robotic cargo spacecraft used to transport supplies to the International Space Station. We refer the reader to two canonical references [28, 56] for further technical background on abstract interpretation.

Zero-knowledge proofs. The notion of zero-knowledge proofs was first proposed in the early work of Goldwasser et al. in [39]. Theoretical constructions based on probabilistically checkable proofs were introduced in the seminal work of Kilian [49] and Micali [54]. In recent years there is significant progress in constructing efficient ZKP protocols that can be realized in practice. There are ZKP schemes based on bilinear pairing [17, 25, 27, 37, 42, 53, 57], discrete-log [12, 19, 23, 41], hashing [20], interactive oracle proofs (IOP) [11, 13, 15, 25, 70] and interactive proofs [61, 67, 69, 71, 72]. Their security relies on different assumptions and models, and they provide trade-offs between prover time and proof size. In our construction, we choose to use [42] and [61] as our backend, but our frontend is also compatible with other circuit-based ZKP schemes.

Most ZKP schemes model the computations as arithmetic circuits, while the abstract interpretation algorithms are naturally in the RAM model with dynamic loops, branches and memory operations. Several papers [14, 16, 21, 22, 66, 73] proposed ZKP schemes for RAM programs. These schemes propose universal RAM-to-circuit reductions to compile any RAM program to arithmetic circuits. However, the heavy machinery introduces a high overhead on the size of the circuit. E.g., the cost is around 4000 gates per RAM instruction in the reduction of [16]. Instead, we utilize several key techniques in these schemes, without going through the full RAM-to-circuit reductions. Among these RAM-based ZKP schemes, the scheme in [66] first compiles the RAM program to a program-specific circuit, then applies a circuit-based ZKP backend. In Section 6.2, we compare the performance of our scheme with the circuits automatically generated by the scheme in [66] and show that our design improves the circuit size by around 35–40×.

2 PRELIMINARIES

We use $\mathsf{negl}(\cdot) : \mathbb{N} \to \mathbb{N}$ to denote the negligible function, where for each positive polynomial $f(\cdot)$, $\mathsf{negl}(k) < \frac{1}{f(k)}$ for sufficiently large integer k. Let λ denote the security parameter.

2.1 Zero-knowledge Arguments

A zero knowledge argument scheme is a protocol between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, where at the end of the protocol, $\mathcal{V}$ is convinced by $\mathcal{P}$ that the result of a computation C on a public input x and prover's secret witness w is $y = C(x, w)$. A zero knowledge argument has (1) correctness: $\mathcal{V}$ always accepts if the result and the proof are honestly computed by $\mathcal{P}$; (2) soundness: $\mathcal{V}$ rejects with all but negligible probability if the result is not correctly computed; (3) zero knowledge: the proof leaks no information about the witness w beyond the fact the $C(x, w) = y$. We give the formal definitions of zero knowledge arguments in Appendix A.

2.2 Instantiating Abstract Interpretation

Abstract Interpretation Fundamentals. When applying Abstract Interpretation, one begins with a ground-truth semantics for programs p, commonly encoded as a partial function $\mathcal{S}_p \in \mathsf{state} \rightharpoonup \mathsf{state}$ where $\sigma \in \mathsf{state}$ is some semantic domain for intermediate execution states. For example, the program p could be encoded as map from program labels to statements, and the state of the program could include the current program label to execute, as well as a map from program variables to their current values. This transition function $\mathcal{S}_p$ can be iterated from an initial configuration $\sigma_0 \in \mathsf{state}$ to a final state $\sigma_n \in \mathcal{S}_p^n(\sigma_0)$ (for some number of steps n), and where σ_n is called final iff $\mathcal{S}_p(\sigma_n)$ is not defined.

From this "ground truth" semantics, a collecting semantics is derived $C_p \in \wp(\mathsf{state}) \to \wp(\mathsf{state})$ which re-characterizes the semantics to transform properties (i.e., sets) of states as opposed to discrete states. This collecting semantics establishes a ground truth for asking questions like "when given an initial state where variable x is negative, does the program result in a state where x is positive?" The collecting semantics is a specification and in general is not computable. A program analysis algorithm then inhabits $\mathcal{A}_p \in \mathsf{state}^\sharp \to \mathsf{state}^\sharp$ for some *abstract* semantic domain $\mathsf{state}^\sharp$, and must be shown sound w.r.t. the collecting semantics C_p. The full analysis is computed (naively) as the smallest solution (i.e., the least fixpoint) to the equation $X = X \sqcup \mathcal{A}_p(X) \sqcup \alpha(\sigma_0)$, which soundly approximates (by construction) any final state so long as $\mathcal{A}_p$ is sound. Although analysis algorithms $\mathcal{A}_p$ are typically designed in an ad-hoc manner, the abstract interpretation framework can also be used to systematically derive the analysis algorithm itself, yielding an analyzer which is correct by-construction; this is often referred to as the calculational method [28].

The analysis algorithm $\mathcal{A}$ must operate over an *abstract domain*—often annotated with a "sharp" ($^\sharp$)—and must form an order-theoretic lattice in addition to a *Galois connection* with sets of concrete states. Computability of the analysis—i.e., the least fixpoint of the equation $X = X \sqcup \mathcal{A}_p(X) \sqcup \alpha(\sigma_0)$—is predicated on either the abstract domain being finite, or the use of a widening operator ∇ to ensure there are no "infinite ascending chains" while computing the analysis.

In general, a Galois connection between lattices A—called the concrete domain—and B—called the abstract domain—consist of two mappings $\alpha \in A \to B$—called the abstraction function—and $\gamma \in B \to A$—called the concretization function—such that $x \sqsubseteq_A \gamma(y) \iff \alpha(x) \sqsubseteq_B y$; Galois connections are notated $A \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} B$. When the context is unambiguous, we omit subscripts to abstraction and concretization functions α and γ, partial ordering $\sqsubseteq$, and lattice operations $\sqcup$. The Galois connection for $\mathsf{state}^\sharp$ is then notated $\wp(\mathsf{state}) \underset{\alpha_{\mathsf{state}}}{\overset{\gamma_{\mathsf{state}}}{\leftrightarrows}} \mathsf{state}^\sharp$, and the correspondence becomes $R \subseteq \gamma(r^\sharp) \iff \alpha(R) \sqsubseteq r^\sharp$ where $R \in \wp(\mathsf{state})$, $r^\sharp \in \mathsf{state}^\sharp$, the lattice over the concrete domain is the powerset lattice, and $\sqsubseteq$ is a custom-defined partial order for the lattice associated with the abstract domain $\mathsf{state}^\sharp$. Abstract states $\mathsf{state}^\sharp$ often include an environment $\mathsf{env}^\sharp \triangleq \mathsf{var} \to \mathsf{val}^\sharp$ mapping program variables to values, and an abstract domain for values $\mathsf{val}^\sharp$ such that $\wp(\mathsf{val}) \underset{\alpha_{\mathsf{val}}}{\overset{\gamma_{\mathsf{val}}}{\leftrightarrows}} \mathsf{val}^\sharp$. A program analysis algorithm $\mathcal{A}$ is then sound when any of the following equivalent (under assumption of Galois connection laws) statements are true: (1): $\alpha(C_p(\gamma(r^\sharp))) \sqsubseteq \mathcal{A}_p(r^\sharp)$; (2): $C_p(\gamma(r^\sharp))) \subseteq \gamma(\mathcal{A}_p(r^\sharp))$; (3): $\alpha(C_p(R)) \sqsubseteq \mathcal{A}_p(\alpha(R)$; or (4): $C_p(R) \subseteq \gamma(\mathcal{A}_p(\alpha(R)))$.

Forming a Galois connection isn't strictly necessary for soundness—it merely shows that the abstract domain is "tight". Calculational methods make great use of the Galois connection, and completing a soundness argument using only the γ side of the connection (the second statement above) is commonly referred to as the "γ-only" framework [58].

Soundness means that the behavior of any concrete run of the program is guaranteed to be approximated by the analysis results. Verification is achieved when the analysis result does not contain any unwanted behavior, i.e., if the analysis result does not include, say, buffer overflows, then we know it is impossible for a buffer overflow to occur when running the program. However, false positives are possible: if the analysis result says the result may contain a buffer overflow, there is a possibility that the program is free of buffer overflows, but the analysis is not precise enough to detect it.

Our Instantiation of Abstract Interpretation. In this work, we implement in zero-knowledge the following instantiation of the abstract interpretation framework: a small imperative language with conditionals and while loops, as well as an extension to the language which includes toplevel functions; a worklist-based flow-sensitive analysis algorithm which relies on control flow information; and instantiations of the worklist-based algorithm for interval analysis, taint analysis and control flow analysis.

As noted previously, analysis results are naively computed as the least fixpoint of the equation $X = X \sqcup \mathcal{A}_p(X) \sqcup \alpha(\sigma_0)$, using the abstract transfer function $\mathcal{A}_p$, initial state σ_0, and abstraction function α. There are two practical matters missing from this simple characterization of computing analysis results. First, there are several orthogonal notations of "sensitivity" in program analysis, such as flow sensitivity, object sensitivity, context sensitivity, etc.. Incorporating each of these vectors typically amounts to appropriately altering the definition of state$^\sharp$ to include extra information, yielding more precision at the expense of a more computationally costly analysis. Second, when implemented directly, this computation is overly naive, e.g., this approach may re-analyze the entire program multiple times when its only necessary to re-analyze a few statements. As for the first practical matter, most analyses adopt at the very least flow sensitivity, as the precision gained vs efficiency lost is nearly always desirable. As for the second practical matter, most analyses are split into two phases: (1) compute a control-flow graph for statements in the program, and (2) using this control-flow graph, compute the analysis property of interest.

We therefore adopt a *flow-sensitive* and *control-flow worklist-based* algorithm, displayed as Algorithm 1, and which we have drawn nearly directly from a standard textbook on program analysis design by abstract interpretation [56]. The algorithm operates conceptually over an abstract domain of states defined state$^\sharp \triangleq \wp(\text{loc}) \times (\text{loc} \rightarrow \text{env}^\sharp)$ where abstract environments are defined $\text{env}^\sharp \triangleq \text{var} \rightarrow \text{val}^\sharp$ and for some abstract domain of values val$^\sharp$. That is, at any given state of abstract execution, there is a set of program locations to execute next, and a global matrix of abstract values for each pair of program location and program variable. We adopt a set-based notation for the function space $S = \{s_l\}_{l=1}^n \in \text{loc} \rightarrow \text{env}^\sharp$, and its applications $s_l \in \text{env}^\sharp$ for the abstract environment at location l.

A common optimization is to split this (hypothetical) transfer function $\mathcal{A}_p \in \text{state}^\sharp \rightarrow \text{state}^\sharp$ in two: $\text{CFG}_p \in \text{loc} \rightarrow \wp(\text{loc})$ as a statically determined over-approximation of control flows, and $\mathcal{A}_{p,l} \in \text{loc} \times \text{env}^\sharp \rightarrow \text{env}^\sharp$ as an update to the abstract environment

Algorithm 1 Worklist Algorithm

Input: A program p, control flow graph CFG_p, transfer function $\mathcal{A}_{p,l}$, and lattice val$^\sharp$.
Output: Abstract environment at each line $\{s_l\}_{l=1}^n$.

```
1: Init $s_l(x) := \bot_{\text{val}^\sharp}$ for all $l$ and $x$.
2: Init queue: $W := \{(l, l') \mid l' \in \text{CFG}_p(l)\}$.
3: while $W \neq \varnothing$ do
4:     $(l, l') = W.pop()$
5:     if $\mathcal{A}_{p,l}(s_l) \not\sqsubseteq s_{l'}$ then
6:         $s_{l'} = s_{l'} \sqcup \mathcal{A}_{p,l}(s_l)$
7:         for all $l''$ follows $l'$ do
8:             $W.push(l', l'')$
9: return $S = \{s_l\}_{l=1}^n$
```

env$^\sharp$ which (abstractly) interprets the program at location l; in the analysis algorithm, $\mathcal{A}_{p,l}$ is always applied to the environment s_l. Note that CFG_p is easily lifted to sets of locations $L \in \wp(\text{loc})$ via $\bigcup_{l \in L} \text{CFG}_p(l)$. The final analysis result $\{s_l\}_{l=1}^n$ is then recursively specified as $s_l^{\text{fin}} = \bigsqcup_{l' | l \in \text{CFG}_p(l')} \mathcal{A}_{p,l'}(s_{l'})$; the algorithm we implement efficiently computes the smallest solution to the set of recursive equations generated by s_l^{fin} for each program location l.

Concretely, Algorithm 1 takes as input a control flow graph of the program CFG_p and initializes a control flow queue $W \in \wp(\text{loc} \times \text{loc})$ as $W = \{(l, l') \mid l' \in \text{CFG}_p(l)\}$, i.e., all control flows. The abstract environment is initialized as $s_l(x) := \bot$ for all program variables x and for the bottom element of the abstract value lattice $\bot$. For a straightline program, W has a simple definition mapping locations to their successor: $W(l) \triangleq \{l+1\}$. For a program with a while loop, W will associate the line before the loop to both the first line of the loop and the first line after the loop, and associate the last line of the loop with the first line of the loop and the first line after the loop. For programs with functions, W encodes the control flow graph of function call and return edges.

The algorithm iterates over all control flows using a queue, updating $s_{l'}$ to $s_{l'} \sqcup \mathcal{A}_{p,l}(s_l)$ (line 6); this updates the analysis results at location l' to include new information, e.g., due to a new analysis result at location l. If the update leads to a new value for $s_{l'}$, then we must re-analyze all program points which are influenced by program point l', so we add flows (l', l'') to the worklist for $l'' \in \text{CFG}_p(l')$ (lines 5, 7 and 8). When instantiated to abstract domains with infinite ascending chains (such as interval analysis), the join operation $\sqcup$ in Algorithm 1 at line 6 is replaced with a widening operator ∇ which approximates $\sqcup$ while avoiding infinite ascending chains. In practice, and for interval analysis, this amounts to approximating $([1,2] \sqcup [1,3]) \sqcup [1,4] = [1,3] \sqcup [1,4] = [1,4]$ as $([1,2] \;\nabla\; [1,3]) \;\nabla\; [1,4] = [1,\infty] \;\nabla\; [1,4] = [1,\infty]$, i.e., it forces early convergence of the upper bound to ∞.

Unique Properties of Abstract Interpretation for ZK. Many applications of zero-knowledge are able to exploit structural or algebraic properties of algorithms to achieve efficiency gains. In general, such insights arise from the observation that *checking* $y = f(x)$ need not always *compute* $f(x)$, e.g., one can instead check $f^{-1}(y) \ni x$. The framework of abstract interpretation prescribes an algorithmic approach based on computing least fixpoints of recursively defined equations, such as $s_l^{\text{fin}} = \bigsqcup_{l' | l \in \text{CFG}_p(l')} \mathcal{A}_{p,l'}(s_{l'})$ in the case of our particular worklist algorithm, or $X = X \sqcup \mathcal{A}_p(X) \sqcup \alpha(\sigma_0)$ in general. Computing the least solution is computationally expensive, and encoded as an iterative process, starting from the $\bot$ element of the lattice, and applying the equations successively until the solution stabilizes on a result.

However, *checking* that a proposed solution is *larger or equal* to the least fixpoint is simple and can be computed in one step without any iteration. This insight applies to zero-knowledge applications of abstract interpretation where the prover wants to show the absence of bugs in a secret program: it suffices to present *any* solution to the equation (it need not be the smallest) so long as the solution demonstrates the absence of bugs. The prover will likely discover this solution through the iterative *least* fixpoint algorithm, but they can do this privately and without the use of cryptography.

On the other hand, if the prover wants to show that a program analysis flags a secret program as potentially having a bug, the entire least fixpoint computation must be executed in zero-knowledge. That is, to be convinced that the program has a bug, the verifier must be convinced that the best possible analysis of the program (i.e., the least fixpoint) is insufficient to verify the absence of bugs. Because no better solution exist (a property of the least solution), the verifier knows the prover did not present an unnecessarily over-approximate result.

Most applications of zero-knowledge abstract interpretation will likely be of the first form, where the prover wants to demonstrate the absence of bugs in a program, and the verifier will be satisfied by merely checking the solution is some fixpoint, and not necessarily the least one. However, the algorithm to just check the solution to the fixpoint—rather than compute it—is realized easily as a small modification to Algorithm 1. Therefore, because the checking algorithm is a simple refinement of the fixpoint-finding algorithm (in both application and algorithmic description) we focus on the full fixpoint-finding algorithm throughout this paper for the sake of completeness. We discuss the implementation-specific details of the refined algorithm and its tradeoffs in Appendix 5.

3 ZERO-KNOWLEDGE ABSTRACT INTERPRETATION

Motivated by the applications mentioned in the introduction, we are working in the following model. The prover owns a secret program. She commits to the program at first, and then later engage in a zero-knowledge proof to convince the verifier the presence or absence of some bugs in the program via abstract interpretation. We assume that the algorithm of the static analysis is public to both the prover and the verifier. The verifier is able to validate the correctness of the result of the analysis through zero knowledge abstract interpretation, while the program of the prover remains confidential during the protocol.

Formally speaking, let $\mathbb{F}$ be a finite field, p be a secret program with n lines and m flows. Suppose the prover and the verifier agree on an abstract interpretation, which is described by a lattice $\mathrm{val}^\sharp$, transfer functions $\mathcal{A}_{p,l}$, a worklist algorithm Alg, and a final calculation g. $g(\mathsf{Alg}(p, \mathrm{val}^\sharp, \mathcal{A}_{p,l})) \in \{0, 1\}$ indicates the presence or absence of bugs. A zero-knowledge abstract interpretation (zkAI) scheme consists of algorithms:

- $\mathsf{pp} \leftarrow \mathsf{zkAI}.\mathcal{G}(1^\lambda)$: given the security parameter, the algorithm generates the public parameter pp.
- $\mathsf{com}_p \leftarrow \mathsf{zkAI.Commit}(p, \mathsf{pp})$: the algorithm commits to the secret program p. We omit the randomness here.
- $(y, \pi) \leftarrow \mathsf{zkAI}.\mathcal{P}(p, (\mathrm{val}^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), \mathsf{pp})$: the prover runs Alg over p to get a sound analysis result S, runs g over S to obtain the result y of the analysis, and generates the corresponding proof π.
- $\{1, 0\} \leftarrow \mathsf{zkAI}.\mathcal{V}(\mathsf{com}_p, (\mathrm{val}^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), y, \pi, \mathsf{pp})$: the verifier validates the claim about the program with parameters of the analysis $(\mathrm{val}^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g)$, and the proof π.

A zkAI scheme also has the properties of correctness, soundness and zero-knowledge as generic zero-knowledge proofs. We give the formal definitions in Appendix B.

3.1 Program Representation

The general idea to construct a zero-knowledge abstract interpretation scheme is as follows. In the beginning, the prover sends the committment com_p of a program p to the verifier. After receiving $(\mathrm{val}^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g)$ from the verifier, the prover computes the analysis result S and the corresponding witness w for proving $g(\mathsf{Alg}(p, \mathrm{val}^\sharp, \mathcal{A}_{p,l})) = 1$. We treat it as some relationship $\mathcal{R} = ((com_p, \mathrm{val}^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g); w)$ in Definition 1. Then the verifier and the prover invoke the backend zero-knowledge proofs protocol to verify the relationship $\mathcal{R}$ without leaking any information of p.

At the beginning of our zkAI scheme, in order for the prover to commit to the secret program, we need an arithmetic representation of the program. Therefore, we first introduce our programming language to work on, and then describe its arithmetic representation. We choose to work on a simple language used in [56], instead of ones such as Java or C++ to avoid language-specific details. However, our zero-knowledge proof scheme does not lose generality because the language is still expressive enough. It is Turing-complete, and languages such as Java and C++ can be compiled to it.

Our Programming Language. We choose to start with a specific imperative programming language as shown in Figure 1. In this language, a program is composed of many statements, and each statement can be either an assignment, a branch, or a loop[2]. a denotes an expression, and it may be a variable x, a constant n, a unary expression op_1 and a binary expression op_2. The value of a variable can be either integers or booleans, and thus we use logical operators $and|or|not$ and mathematical operators $+|-|*|/$ in our language. We will extend this to incorporate function calls in Section 4 as they are common in programming languages even though doing so will not change the expressiveness.

Arithmetic Representation. Then we describe how to convert the program to an arithmetic representation. We choose to use table structures to represent the whole program. Details are shown in Table 1. For each type of statements,

(1) A unique 'Stmt Code' is assigned.
(2) The 'Line No.' field stores the line number of the statement in the program. For `if` and `while` statements, this refers to the line where `if` and `while` condition lies.We treat `else`

[2]In our implementation, we also have statements of memory read and memory write. They are processed similarly to assignments in the worklist algorithm, and we omit them in the description for simplicity.

Stmt	Stmt Code	Line No.	a	Line No.(else)	Line No.(end)	variable ID (x)
x=a	1			/	/	
`if … else … end`	2		/			
`while… do … end`	3		/	/		

Expression a	Expression Code	Variable ID (x_1)	Variable ID (x_2)	Value	Op code
x_1	1		/	/	/
n	2	/	/		/
op_1 x	3		/	/	
x_1 op_2 x_2	4			/	

Table 1: Arithmetic representation of our programs.

$$
\begin{aligned}
stmt &::= x = a \\
&\quad \mid \ \texttt{if}\ x\ \texttt{then}\ stmt\ \texttt{else}\ stmt\ \texttt{end} \\
&\quad \mid \ \texttt{while}\ x\ \texttt{do}\ stmt\ \texttt{end} \\
a &::= x \mid n \mid op_1\ x \mid x_1\ op_2\ x_2 \\
op_1 &::= \texttt{not} \\
op_2 &::= + \mid - \mid * \mid / \mid \texttt{and} \mid \texttt{or}
\end{aligned}
$$

Figure 1: Our imperative programming language.

and `end` as a single line so that they have Line No. as well. This helps upper-bound the number of outgoing edges in later control flow graph construction.

(3) The field a is only used in the assignment expression.

(4) 'Variable ID' refers to the left part of the assignment statements and conditions in `if` and `while` statements.

Similarly, for each expression a, we use 'Expression Code' to identify the type of the expression. We store two possible 'Variable ID's, a 'constant' and an 'Op code'. A '/' symbol in the table means that the field is not applicable for this type of statement or expression and is left empty.

Using these table structures, we can represent the whole program as a sequence of elements in the field $\mathbb{F}$, and the prover can commit it using existing commitment schemes.

3.2 Proving Intra-procedure Analysis

With our simple programming language and its arithmetic representation, one can simply construct a zero-knowledge abstract interpretation scheme using generic zero-knowledge proofs. The prover commits to the secret program, and then proves the result of Algorithm 1 on the program using generic ZKP. Unfortunately, such a naive approach would introduce a high overhead on the efficiency. Most generic ZKP schemes represent the computation as an arithmetic circuit, and turning Algorithm 1 into an arithmetic circuit naively would introduce a high overhead on the size of the circuit. There exist RAM-based ZKP schemes that reduce RAM programs to arithmetic circuits through RAM-to-circuit reduction. The size of the circuit preserves the asymptotic running time of the RAM program. However, the concrete overhead of these schemes in practice is usually high. For example, each RAM instruction costs thousands of arithmetic gates to implement [14, 16, 73].

In this section, we present our construction of the zero-knowledge abstract interpretation. Using ideas in the literature of RAM-based ZKP [14, 16, 21, 22, 66, 73], we construct an efficient arithmetic circuit to validate the correct execution of Algorithm 1 on the committed program. The prover and the verifier then invoke a circuit-based ZKP scheme to prove and validate the result of the abstract interpretation.

Figure 2 shows the main construction of our scheme. The red part is the secret program owned by the prover, the blue part is additional auxiliary input from the prover to validate the abstract interpretation efficiently (see below), and the green part is the result of the analysis.

Based on the functionality, our scheme is mainly divided into four parts: 1) Checking the consistency between the program and the control flow graph. 2) Checking the correct execution of each iteration. 3) Checking lattice operations and transfer functions. 4) Deciding whether there are bugs or not based on the result of Algorithm 1. The first and the second part is problem-independent, and the third and the fourth part is problem-dependent. The fourth part is usually simple in practice, containing only a few conditions. Therefore, we focus on the first three parts in this section.

Control flow graph consistency. Since abstract interpretation works on flows (l, l'), in our zero-knowledge abstract interpretation scheme, we first ask the prover to provide a control flow graph as an auxiliary input to the circuit. In this graph, each node is a line of code labeled by its line number, and an edge denotes a flow (l, l') from node l to node l'. Since we treat `else` and `end` as separate lines, in our simple programming language a node in this graph can have at most two outgoing edges in the case of loops and branches. Therefore, we represent the entire control flow graph using a table of size $n \times 2$. The l-th row of the table stores the target line numbers of the two possible outgoing edges of line l.

We then have to check that this control flow graph is indeed consistent with the program. Note that we can obtain all the possible flows (l, l') from the program representation in Table 1. We go through the whole program line by line, and each statement individually contributes to a few flows. An assignment statement produces only a single flow $(l, l+1)$; a `while` statement produces three flows, $(l, l+1)$, $(l, l_{\texttt{end}}+1)$, $(l_{\texttt{end}}, l)$; an `if` statement produces four flows, $(l, l+1)$, $(l, l_{\texttt{else}}+1)$, $(l_{\texttt{else}}, l_{\texttt{end}})$, $(l_{\texttt{end}}, l_{\texttt{end}}+1)$. The flows deduced by the $n \times 2$ table are simply permutations of these flows obtained by the linear scan of the program, ordered by the first line number l.

In order to check that they are consistent, we apply existing techniques in [61, 67, 73] for testing set equality. In these techniques, we first combine the pair of values in each element of the sets through $\mathcal{H}(l, l') = nl + l'$, where n is the number of lines in the program. Then we compute the characteristic functions h of the sets: $h_A(x) = \prod_{(l,l') \in A}(\mathcal{H}(l, l') - x)$. Once the characteristic functions for the two sets are calculated respectively, a random challenge r from the verifier is generated. If the two characteristic functions

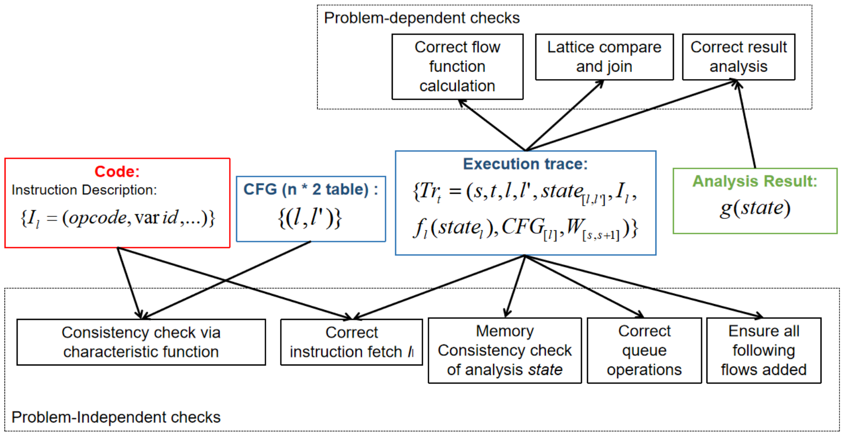

Problem-dependent checks
Correct flow function calculation
Lattice compare and join
Correct result analysis
Code:
Instruction Description:
$\{I_l = (opcode, \text{var}\, id, \ldots)\}$
CFG (n * 2 table) :
$\{(l, l')\}$
Execution trace:
$\{Tr_t = (s, t, l, l', state_{[l,l']}, I_l, f_l(state_l), CFG_{[l]}, W_{[s,s+1]})\}$
Analysis Result:
$g(state)$
Consistency check via characteristic function
Correct instruction fetch I_l
Memory Consistency check of analysis *state*
Correct queue operations
Ensure all following flows added
Problem-Independent checks

Figure 2: Our circuit checking the correctness of the abstract interpretation in Algorithm 1.

agree on the random point r, then they will be the same with overwhelming probability by the Schwartz-Zippel Lemma [60, 74]. The size of the circuit for the check above is $O(n)$.

Correct execution of the iteration. With the control flow graph provided by the prover, we then show how to execute each iteration of Algorithm 1. The correct execution of the iteration can be further divided into executing queue operations, fetching instructions, reading and writing analysis states and extracting following flows.

The queue operation is not naturally supported by circuits. A queue W has *push* and *pop* operations, which follows a first-in-first-out strategy. We model the queue as an array arr with a head index s and a tail index t. When $W.push(e)$ is called, we increment t, and write the element e into $arr[t]$. Similarly, when $W.pop()$ is called, we return $arr[s]$ and then increment s. For the queue W used in Algorithm 1, we observe that exactly one flow is popped in each iteration and at most 2 flows are pushed into the queue, as each line has at most two outgoing flows. Therefore, we can ask the prover to provide the entire execution trace of the queue W as an auxiliary input. The circuit first checks that the queue is initialized correctly, i.e., the first m flows are the same as those in the control flow graph. Then in each iteration, s increases by 1 and the pop operation in step 4 is a linear scan over the entire trace.

t, however, is more complicated to deal with. As at most 2 flows are pushed in each iteration, we ask the prover to provide $arr[t]$ and $arr[t+1]$ as auxiliary input. The circuit then performs the computation of Step 5-8 in Algorithm 1. This is the combination of several cases: if the condition in Step 5 is false, the circuit does nothing and keeps t unchanged; otherwise, the circuit compares $arr[t]$ and $arr[t+1]$ to the flows of l' in the control flow graph. If l' has only one flow, we only compare it with $arr[t]$ and set $t = t + 1$; otherwise we compare the two flows with $arr[t], arr[t+1]$ and set $t = t + 2$. How to compute the condition of Step 5 and fetch the flows of l' are explained later. In this way, the circuit to deal with the queue operation for each iteration is of constant size. Finally, we need to make sure that the auxiliary input $arr[t], arr[t+1]$ in every iteration is consistent with the entire trace of the queue. This cannot be done by a linear scan, as t increases by 0, 1 or 2 in different cases. Instead, we view them as memory read at address t and $t + 1$ from the trace of the queue and validate their correctness by memory consistency checks, which we describe later. We will use the memory checking techniques extensively below.

The second component in each iteration is the instruction fetch. As shown in Algorithm 1, a flow (l, l') is popped from queue W in step 4 and we need to fetch line l from the program committed by the prover in order to determine the transfer function $\mathcal{A}_{p,l}$ and the IDs of the variables touched by line l. If we view the program as a memory indexed by the line number, this instruction fetch is a classical read operation of a random access memory. Thus we ask the prover to provide the expected content of line l in the program, and validate its correctness by memory consistency checks.

The third component is updating the states of variables in step 5 and 6. As explained in Section 2.2, $S = \{s_l\}_n$ and each s_l consists of the states of all variables. Thus we use an $n \times v$ matrix to represent S, where v is the total number of variables in the program. Every value in the matrix is one of the states in the Lattice val$^\sharp$, initialized to $\perp$ (also mapped to a particular value in the field). In step 5 of Algorithm 1, we perform memory read at l to obtain the states of variables used by $\mathcal{A}_{p,l}$; in step 6, if step 5 is true, we perform memory write to l' to update the states of variables in $s_{l'}$.

Another operation in step 7 is to extract all lines following l' if the states $s_{l'}$ is updated. Thanks to our representation of programs, one line can have at most two following lines in the intra-procedural analysis. Thus this operation is a memory read from the control flow graph represented as an $n \times 2$ table. We let the prover provide the content of line l' in the table, and launch another memory consistency check to ensure its correctness. The flows are then compared to $arr[t]$ and $arr[t+1]$ as explained above for the push operation in the queue.

Memory consistency check. We used the memory consistency check heavily in the design above, and here we described our techniques in details. Memory checking is commonly used in RAM-based zero-knowledge proof schemes to validate the correctness of memory accesses by a circuit efficiently. To check T memory accesses, the circuit size is $O(T \text{polylog}(T))$, instead of $O(MT)$ naively where M is the size of the entire memory. Ben-Sasson et al. [14] introduced a memory checking scheme using permutation networks, which is later refined in [16, 21, 73]. In this paper, we use a memory checking scheme called *offline memory checking*, introduced recently in [61, 62] based on the idea of the earlier work [18].

Offline memory checking. Formally, we view the memory as a vector of key value pairs $(k, \mathbf{v})$, where $\mathbf{v}$ itself can be a vector of values. Each $\mathsf{Read}(k)$ operation fetches the value $\mathbf{v}$ associated with key k, and each $\mathsf{Write}(k, \mathbf{v}')$ changes the value associated with k to $\mathbf{v}'$.

The offline memory checking technique [18, 62] reduces the correctness of memory operations to a set relationship. In particular, it introduces two timestamps t and ts, a read set RS and a write set WS. ts is the current timestamp, incremented by 1 after every Read or Write operation. t (for a key/address k) is the timestamp when the value at k is accessed last time. Now the memory becomes a vector of tuples $(k, \mathbf{v}, t)$. When performing a memory operation ($\mathsf{Read}(k)$ or $\mathsf{Write}(k, \mathbf{v}')$),

- Fetch the tuple $(k, \mathbf{v}, t)$ from the memory and add it to RS,
- Update the current timestamp as $ts = ts + 1$,
- For a Read, add the tuple $(k, \mathbf{v}, ts)$ to WS and update the memory at k as $(k, \mathbf{v}, ts)$; for a Write, add the tuple $(k, \mathbf{v}', ts)$ to WS and update the memory at k as $(k, \mathbf{v}', ts)$.

Observe that all the elements in the RS and WS are unique as the timestamps are increasing, and the RS is always lacking behind WS by the last state of the memory. Therefore, it is shown that the memory is correctly executed if and only if $\mathsf{Init} \cup WS = RS \cup \mathsf{Final}$, where Init and Final denote sets representing the initial state and the final state of the memory.

Checking memory consistency in circuits. In our zero-knowledge abstract interpretation scheme, all information regarding the memory operations are provided by the untrusted prover. Therefore, it requires additional input and checks to perform the offline memory checking. Taking the instruction fetch as an example. We view the instructions in the secret program as a read-only memory. The secret program was already committed by the prover and it is not hard to check that the program is well-formed in our programming language. Therefore, the initial state of the memory is well-defined. During the execution of the worklist algorithm, the current timestamp ts is simply the numbering of the current iteration. We add a counter starting from 0 and increase it by 1 in each iteration, thus ts is always correctly computed. As the instructions are read-only, in every instruction fetch the prover provides the line number l and its value $\mathbf{v}$, and we use them both for the read set RS and the write set WS. The only missing part is the timestamp t when the instruction was last accessed. We ask the prover to further provide t for every read operation as an auxiliary input. In addition, we check that $t \le ts$ in the circuit. With all of the information above, each read operation adds $(l, \mathbf{v}, t)$ to RS and $(l, \mathbf{v}, ts)$ to WS.

Some components in our scheme, such as updating the states of variables, has both memory read and memory write operations. Similar to the case above, the states of all variables are initialized to all 0s, which defines the initial state of the memory correctly. The timestamp ts can again be correctly computed with the iterations of Algorithm 1. When writing to a memory address with key k and value $\mathbf{v}'$ (computed by the transfer function and the lattice), we add $(k, \mathbf{v}', ts)$ to WS. However, the algorithm never uses the original value $\mathbf{v}$. In this case, we ask the prover to provide both v and the timestamp t as auxiliary inputs. The circuit again checks that $t \le ts$ in every memory operation. We show in Appendix C that with this additional check of $t \le ts$ in both cases ensures the correctness of the memory operations.

Set relationship. To complete our memory consistency check, we again rely on the characteristic polynomials of the sets, as in [21, 61, 62]. We first compress the tuple by a random linear combination $\mathcal{H}(k, \mathbf{v}, t) = k + r \cdot t + \sum_i r^{i+2} v_i$, where r is randomly chosen by the verifier. Then the characteristic polynomial of a set A is $h_A(x) = \prod_{a \in A}(\mathcal{H}(a) - x)$. The circuit computes the characteristic polynomials $h_{RS}(x), h_{WS}(x), h_{\mathsf{Init}}(x), h_{\mathsf{Final}}(x)$, and checks that $h_{WS}(\gamma) \cdot h_{\mathsf{Init}}(\gamma) = h_{RS}(\gamma) \cdot h_{\mathsf{Final}}(\gamma)$, for a random γ chosen by the verifier. This guarantees that $\mathsf{Init} \cup WS = RS \cup \mathsf{Final}$ with overwhelming probability by the Schwarz-Zippel lemma.

The circuit size of our memory checking technique is $O(T \log T)$ for T memory operations. Note that the schemes in [61, 62] achieve linear complexity as the verifier knows the memory access pattern. In our case, all the information are provided by the prover and validated in the circuit, thus these schemes are not sufficient. Comparing to existing memory checking techniques in [14, 16, 21, 73], though our complexity is asymptotically the same, concretely in our scheme the circuit only checks $t \le ts$ in each memory write, while the existing schemes check both the memory addresses and the timestamps are sorted. Our circuit is smaller in practice, and our memory checking scheme may be of independent interest.

Lattice operations and transfer functions. Lattice operations and transfer functions in step 5 and 6 of Algorithm 1 are problem-dependent. Generally speaking, after fetching the states of variables from s_l, we implement the circuit to compute the transfer function $\mathcal{A}_{p,l}$ on the state of each variable, denoted as $s_l^* = \mathcal{A}_{p,l}(s_l)$. Then for every variable, we implement the compare operation on s^*l and s_l' and outputs 1 if $s^*l \not\sqsubseteq s_{l'}$. As the lattice $\mathsf{val}^\sharp$ is a partially ordered set, the circuit size for the compare function is quadratic in the number of states in $\mathsf{val}^\sharp$ in the worst case. Finally, if the compare function outputs 1 for any variable, we implement the join operation of the lattice $s_{l'} = s_{l'} \sqcup s_l^*$ in step 6 in the circuit. The size of the circuit in this step varies a lot for different analyses. In this paper, we focus on specific ones illustrated below. Compiling these functions to circuits automatically and efficiently is left as an interesting future work.

Here we give two examples we use in the experiments later: tainting analysis and interval analysis. For the tainting analysis, the lattice is small and finite, consisting of only two values: UnTainted and Tainted. The transfer function monitors the flow of tainting information. For statements of assignment such as $a = x_1\ op\ x_2$, it sets a to Tainted if either x_1 or x_2 is Tainted. Statements of `If ... else` and `While` do not change the state at all. Looking ahead, for inter-procedure analysis with function calls in Section 4, depending on the applications some functions are defined as tainting sources, sanitizers or safe procedures. When seeing these function calls, the transfer function sets the state of the variable to Tainted, UnTainted or same as the input respectively. The compare and the join operations are also very simple. We define UnTainted < Tainted, and UnTainted $\cup$ Tainted = Tainted in the lattice. Therefore, in tainting analysis, the transfer function, the compare and the join operation can be implemented as circuits of constant size.

For the interval analysis, the lattice has an infinite size. It consists of all the intervals of the form $[l, r]$, where l and r are integers in most cases, and can be ∞ and $-\infty$ as well to denote uncertainty. The transfer function computes the possible range of variables based on the instructions and the input. For example, an instruction $a = 2$ changes the state of a to $[2, 2]$; an instruction $a = x_1 + x_2$ with input states $x_1 = [l_1, r_1], x_2 = [l_2, r_2]$ set the state of a to $[l_1 + l_2, r_1 + r_2]$; an instruction $a = x_1 - x_2$ with input states $x_1 = [l_1, r_1], x_2 = [l_2, r_2]$ set the state of a to $[l_1 - r_2, r_1 - l_2]$. The compare operation is defined as the subset relationship of intervals, i.e., $[l_1, r_1] \subseteq [l_2, r_2]$ if $l1 \geq l_2$ and $r_1 \leq r_2$, which is a partial ordering. The join operation returns the tightest interval that contains the union of the two intervals, i.e., $[l_1, r_1] \cup [l_2, r_2] = [\min(l_1, l_2), \max(r_1, r_2)]$. The size of circuits for these functions is linear to the bit-length of the integers.

Widening. Our scheme also supports the widening operator ∇ with a small overhead. We add a counter to each line of code initialized to 0 and increase it by 1 every time the line is analyzed in the worklist algorithm. When the counter reaches the predefined threshold, the algorithm forces the early convergence, i.e., setting the bounds to ∞ and $-\infty$ in the interval analysis. Updating the counter for each line of code also consists of memory accesses, thus it is convenient to store it together with the states of the variables in an $n \times (v + 1)$ table, and slightly modify the transfer function to include the case above. This approach only introduces a small overhead on the size of the entire circuit.

Complexity. Overall, with all the building blocks explained above, the total size of the circuits for the examples we consider is $O(T \cdot v + T \log T)$, where T is the number of iterations of the worklist algorithm and v is the number of variables. The first term $O(T \cdot v)$ hides the complexity of transfer functions, compare and join operations depending on different analyses. This is asymptotically the same as the plain worklist algorithm in Algorithm 1 up to a logarithmic factor. In practice, we observe that the first term updating the states of all variables is the dominating part.

3.3 Putting Everything Together

After reducing the correct execution of the worklist algorithm to the circuit in Figure 2, we then apply the a generic zero knowledge proof scheme as the backend on the circuit and complete the construction of our zero-knowledge abstract interpretation scheme. The formal protocol is described in Protocol 1 in Appendix D. We have the following theorem:

Theorem 1. *Protocol 1 is a zero-knowledge abstract interpretation scheme by Definition 2.*

The theorem follows the correct reduction to the circuit in Figure 2, the correctness, soundness and zero-knowledge of the backend. We give a proof sketch in Appendix D.

Complexity. The overall complexity depends on the backend of zero knowledge proof scheme. For example, using the pairing-based SNARK [42], the prover time is $O(T \cdot v \log T + T \log^2 T)$, the proof size is $O(1)$ and the verifier time is $O(1)$, where T is the number of iterations of the worklist algorithm and v is the number of variables; using the ZKP in [61], both the prover time and the verifier time are $O(T \cdot v + T \log T)$, and the proof size is $O(\sqrt{T \cdot v + T \log T})$.

4 PROVING INTER-PROCEDURAL ANALYSIS

In this section, we show how to extend our construction of zero-knowledge abstract interpretation in Section 3 to inter-procedural analysis for programs with function calls.

```
fdef ::= fname(x, ..., x) begin stmt ... stmt end
stmt ::= ... | x = fname(x, ..., x)
```

Figure 3: Function calls in our programming language.

4.1 Inter-procedural Abstract Interpretation

Inter-procedural abstract interpretation is more demanding because it takes function definitions and function calls into account. Complications arise when dealing with the mechanism of arguments. Generally speaking, at the beginning of a function call, the program switches into a new variable scope, and pass all the arguments from the caller's variable scope to the new one. At the end of a function call, the return value is passed back from callee's variable scope.

Algorithm 1 works on flows and transfer functions, so it is still possible to use it for inter-procedural analysis as long as the control flows introduced by function calls and the corresponding transfer functions are properly designed.

To properly modify the control flows, a few additional structural instructions are added to the program. At the definition of a function p, an instruction $init(p)$ is used to mark the beginning of a function, and an instruction $final(p)$ is used to mark the end. When calling function p, the call statement is split into two statements, i.e. $enter(p)$ and $exit(p)$. Three additional flows between these instructions are added to deal with function calls, i.e. $enter(p) \longrightarrow init(p)$, $final(p) \longrightarrow exit(p)$, and $enter(p) \longrightarrow exit(p)$. To ensure that arguments and return values are passed properly, the transfer functions located at the entrance and exit of function calls pass in the arguments and pass out the return value, respectively.[3]

4.2 Modifications to Our zkAI Scheme

To incorporate the changes of the inter-procedural abstract interpretation above, we modify our programming language, and address several critical challenges in this section.

Modification to the programming language. First, we add function calls to our programming language, as shown in Figure 3. This is a natural extension of the original programming language in Figure 1. Now a program consists of several function definitions, and each function definition has multiple arguments and statements. For the statements, we introduce another type of function calls in addition to the original statements of assignment, branch and loop. With the inclusion of function definitions and calls to this language, we can conduct an inter-procedural analysis.

Challenges. This additional statement of functions introduces several challenges to our zero-knowledge abstract interpretation scheme. The key reason is that each statement can have more than two variables as the function arguments, and there can be more than two flows going in to and out of a line because of function calls. One could set an upper bound on these and use the same zero-knowledge abstract interpretation in Section 3, but it would introduce an multiplicative overhead of the upper bound. Instead, we present several techniques to reduce the inter-procedural analysis on programs with function calls to a circuit of the optimal size.

[3]We focus on context insensitive analysis in this paper since it is the most common design used in large-scale analysis tools, but our techniques can be extended to support context sensitive analysis.

Algorithm 2 Access Linked List

```
Input: The linked list represented by arrays head, next, data. A node $nd$
Output: data$[n]$ for all nodes $n$ in the list of $nd$.
1: for ($pt$ = head$[nd]$; $pt \neq$ NULL; $pt$ = next$[pt]$) do
2:     Output (data$[pt]$).
3: return
```

Program Representation. We think of a function as a fragment of code. To represent a function definition, we store the start and end line number of the function. Besides, we also store the number and the type of the function arguments. We store all these information in a function definition table.

Then we deal with the statements of function calls. The arithmetic representation in Section 3.1 is efficient for intra-procedural analysis due to the bounded number of fields required to represent a statement. However, this is not the case for inter-procedural analysis because a 'function call' statement requires as many fields as the number of function arguments. As a result, if we use the same arithmetic representation as in Section 3.1, every statement will have the same number of fields as the function with the most arguments, which leads to a large overhead in practice.

Our solution is to create a 'function call' table which contains the arguments of 'function call' statements. Then, in the original 'statement' table, an additional index to the 'function call' table is added. This construction frees normal statements from dummy fields. Since circuits do not naturally support indexing, we add another memory checking whenever accessing these arguments.

Representing the control flow graph as a linked list. With the function calls, each line of code can have more than two incoming or outgoing flows. Thus if we still represent the control flow graph as a matrix, the size would be n by the maximum number of flows from any line. Instead, we propose an approach to simulate the linked list in circuits efficiently using techniques of memory checking.

We construct 3 arrays: head, next and data. head is an array of size n and the l-th element stores the index of the start of the list for l in next and data. next and data are of size m. data$[pt]$ stores the data (flow) at a node pointed by an index pt, and next$[pt]$ stores the index of its next node as in a linked list. We use a special character (e.g., -1 in the field) to denote the end of the list (NULL). With these three arrays, we can traverse the linked list using the simple algorithm in Algorithm 2. To validate the traversal in a circuit, we ask the prover to provide the expected pt and $data[pt]$ in each iteration of Algorithm 2, and check their consistency with array next and data using the memory checking technique in Section 3.2.

For our zero-knowledge abstract interpretation scheme, as shown in Figure 2, we traverse the CFG with all lines to compute all flows and compare it with the flows computed from the program. The size of the circuit above doing so is $O(m)$. In addition, during the worklist algorithm, we fetch all the flows from l' in each iteration of Algorithm 1, and the circuit size for the linked list operations above is optimal (the total number of flows in the worklist algorithm).

Loop Merge. With this linked-list representation of the control flow graph, the only remaining challenge is the worklist algorithm. In particular, since the number of flows from a line l' is not a constant anymore, the number of iterations in the loop of step 7 varies in every iteration of the outer loop. Compiling the algorithm naively to a circuit will again introduce an overhead of the maximum possible number of flows in every iteration. In order to solve this problem, we borrow an idea named *flattening* from prior work in zero-knowledge [66] which merges two (or more) loops into one, and we can bound the number of iterations of the outer loop.

Algorithm 3 Verification of the Worklist Algorithm

```
Input: A program $p$, transfer function $\mathcal{A}_{p,l}$, lattice val$^\sharp$, initial state $W_{init}$, and final state $W_{final}$ of the worklist.
Output: Abstract state at each line $\{s_l\}_n$.
1: Init $s_l(x) = \perp_{\text{val}^\sharp}$ for all $l$ and $x$.
2: The first $m$ flows in $W_{final}$ is the same as $W_{init}$.
3: for $i = 1 \to |W_{final}|$ do
4:     $(l, l') = W_{final}[i]$
5:     if $f_l'(s_l) \not\sqsubseteq s_{l'}$ then
6:         $s_{l'} = s_{l'} \sqcup \mathcal{A}_{p,l}(s_l)$
7:         need_update$[i]$ = True
8:     else
9:         need_update$[i]$ = False
10: $t = m + 1$
11: for $i = 1 \to |W_{final}|$ do
12:     $(l, l') = W_{final}[i]$
13:     if need_update$[i]$ then
14:         for ($pt$ = head$[l']$; $pt \neq$ NULL; $pt$ = next$[pt]$) do
15:             Check $W_{final}[t]$ = data$[pt]$.
16:             $t = t + 1$
```

Algorithm 4 Loop Merged of Algorithm 3 step 10-16

```
1: $t = m + 1$
2: $i = 1$
3: $(l, l') = W_{final}[i]$
4: $pt$ = head$[l']$
5: while $i \leq |W_{final}|$ or $pt \neq$ NULL do
6:     if need_update$[i]$ and $pt \neq$ NULL then
7:         Check $W_{final}[t]$ = data$[pt]$
8:         $t = t + 1$
9:         $pt$ = next$[pt]$
10:    else
11:        $i = i + 1$
12:        $(l, l') = W_{final}[i]$
13:        $pt$ = head$[l']$
```

To better illustrate the problem and the solution, we first rewrite Algorithm 1 to Algorithm 3 which verifies the worklist algorithm instead of computing it. Note that the algorithm takes the entire final state of the worklist after all iterations as an input provided by the prover. It checks that the first m flows are correctly initialized to W_{init}, all flows of the program, then checks that the remaining flows are pushed into the queue properly. Step 1-9 are almost the same as step 1-6 in Algorithm 1 computing the transfer function, compare and join operations, and we introduce an additional array need_update for the sole purpose of explaining the challenge. In step 10-16, the algorithm is checking that for each flow l' in the worklist, the following flows are pushed to the queue. It accesses the flows from l' using a linked list as shown in Algorithm 2. It is not hard to see that the checks in Algorithm 3 pass as long as W_{final} is correctly computed by Algorithm 1.

Now the problem happens in step 10-16 of Algorithm 3. The size of the outer loop is pre-defined, but the number of iterations of the inner loop in step 14 varies, and implementing the algorithm

Algorithm 5 Checking the Validity of a Solution

```
Input: A program $p$, control flow graph CFG$_p$, transfer function $\mathcal{A}_{p,l}$, and lattice val$^\sharp$, an abstract environment $\{s_l\}_{l=1}^n$.
Output: valid or invalid
1: Init queue: $W := \{(l, l') \mid l' \in \text{CFG}_p(l)\}$.
2: while $W \neq \emptyset$ do
3:     $(l, l') = W.pop()$
4:     if $\mathcal{A}_{p,l}(s_l) \not\sqsubseteq s_{l'}$ then
5:         return invalid
6: return valid
```

naively as a circuit introduces a high overhead as explained above. To solve this problem, we change step 10-16 to Algorithm 4. In Algorithm 4, both conditions on need_update$[i]$ and p are merged into a single loop, and the updates on i, t and p are all processed in the loop. The total number of iterations for the large loop in step 5 is pre-defined, i.e. $2|W_{final}| - m$. Implementing the algorithm as a circuit preserves the number of iterations. The only overhead is that in each iteration, the circuit executes statements in both step 6-9 and step 10-13, which slightly increases the size per iteration.

In addition to the worklist algorithm, in our scheme we also apply the loop merge technique to check the consistency of the control flow graph (computing all the flows from the secret program) with the optimal circuit size. The algorithm is more straight forward than the worklist algorithm and we omit the details here.

Complexity. With these modifications, the size of our circuit for inter-procedure analysis remains $O(T \cdot v + T \log T)$, where v now denotes the maximum number of variables in any functions of the program (we view the main program also as a function). The concrete size of the circuit for the inter-procedure analysis is larger compared to the intra-procedure analysis, as there are additional computations of loop merge, linked list operations and copying states of variables when entering to and exiting from functions.

5 PROVING ABSENCE OF BUGS

As mentioned in Section 2.2, when the prover wants to prove the absence of bugs in the secret program, it suffices for the prover to present any fix point of the abstract interpretation. The verifier validates that it is indeed a solution, instead of validating the whole computation of the worklist algorithm. Here we present the algorithm of the validation in Algorithm 5.

As shown in the algorithm, it is enough to check that for every flow (l, l'), the state $s_l^* = \mathcal{A}_{p,l}(s_l)$ is always 'smaller than' $s_{l'}$ in the partial ordering of the lattice. The algorithm does not update the states iteratively, thus no new flow is pushed into the queue W, as in Algorithm 1. When implementing Algorithm 5 in circuits, the queue is static and we do not need to support the push operation using the loop merge technique. We also do not have the join operation anymore, but in practice the overhead of join after the compare operation is small. However, for inter-procedure analysis with function calls, we still need to represent the CFG as a linked list and check its consistency with the program using the techniques in Section 4. The total size of the circuit becomes $O(m \cdot v + m \log m)$.

6 IMPLEMENTATION AND EVALUATIONS

We implement our zero-knowledge abstract interpretation scheme and present the experimental results in this section.

Software. The schemes are implemented in C++. There are around 2,500 lines of code for our frontend to compile the analyses to a rank-1-constraint-system (R1CS). We use the open-source compiler of libsnark [10] to generate R1CS in our frontend.

Choice of backends. As described in Section 3 and 4, our frontend efficiently compiles the static analysis on a program to an R1CS instance, and we can use any generic zero knowledge proof scheme on R1CS as our backend, including [11, 15, 23, 25, 42, 53, 61]. We choose two of them in our implementation with different trade-offs. The first one is the pairing-based SNARK [42] with the change in [24] for commit-and-prove. The scheme has a constant size proof and fast verifier time. However, the prover time is relatively slow ($O(C \log C)$ on an R1CS of size C) and it requires a trusted setup. The second one is the recent scheme from [61] called Spartan. The scheme does not require trusted setup and the prover time faster than [42], but the proof size is $O(\sqrt{C})$ and the verifier time is $O(C)$.

Hardware. We run our experiments on Amazon EC2 c5.9xlarge instances with 72GB of RAM and 3GHz Intel Xeon platinum 8124m virtual core. We report the average running time of 5 executions.

Benchmarks. In this section, we report the performance of our zero-knowledge abstract interpretation for three analyses: tainting analysis, interval analysis and control flow analysis. The analyses are performed on real programs drawn from the public benchmarks for static analysis tools WCET [9] and DroidBench 2.0 [4], and existing artifacts for control flow analysis [8] used as the benchmarks in the recent paper [68] on abstract interpretation. The WCET project contains programs used to evaluate and compare different types of analysis tools, while the DroidBench 2.0 is designated for evaluating the effectiveness of taint-analysis tools for Android applications.

To perform control flow analysis, we make functions first-class citizens in our language, which means they can be referred to by variables and called anonymously. The abstract domain is then defined as subsets of all possible functions in the program. Comparison and join on this domain are defined as the subset relationship and the set union operation respectively. We initialize the control flow graph as all determined flows, and append all possible flows to the worklist when we reach anonymous calls in the analysis.

All the programs in our benchmarks contain function calls. We support inter-procedure analysis with a small overhead using the techniques in Section 4. Therefore, we show the performance of our zero-knowledge inter-procedure analysis scheme in this section.

6.1 Evaluations on Real Programs

We first evaluate our scheme on five real programs. We run the zero-knowledge interval analysis on bubble sort and FFT programs from the WCET benchmark [9], run the zero-knowledge tainting analysis on the PrivateDataLeak program from the DroidBench 2.0 benchmark [4], and run the zero-knowledge control flow analysis on the RSA and Solovay-strassen programs in [8].

We compile these programs to our programming language. For the PrivateDataLeak program, built-in functions such as `getDeviceId`(), `getPassWd`() in the Android development environment do not exist in our language, and we mark them as either tainting source or safe procedures following the description of the benchmarks. Some other procedures such as `sendTextMessage`() are marked as 'sink' where tainting information should not flow into.

	BubbleSort	FFT	PrivateDataLeak	RSA	Solovay-strassen
#lines	63	156	207	161	301
#variables	14	43	27	22	75
Analysis	Interval	Interval	Tainting	Control Flow	Control Flow
#flows	78	181	232	215	374
#iterations	237	681	291	216	375
circuit size	582,048	4,763,263	277,537	255,056	1,235,354
SNARK [42] prover time	43.89s	355.2s	21.63s	20.21s	91.73s
verification time	1.4 ms				
proof size	128 bytes				
Spartan [61] prover time	6.49s	48.9s	3.27s	1.86s	12.8s
verification time	0.640s	6.54s	0.297s	0.208s	1.69s
proof size	48.1 KB	83.1 KB	31.1 KB	30.3 KB	48.8 KB

Table 2: Performance of our zkAI schemes.

The performance of our schemes are shown in Table 2. #variables denotes the maximum number of variables in a function. The number of lines is counted in our language, as one can embed multiple statements in one line in the original language. We also show the number of flows m and the number of iterations T. As shown in the table, it takes 355.2s to generate a proof for zero knowledge interval analysis on the FFT program with 156 lines of code, 21.63s for zero knowledge taint analysis on the PrivateDataLeak with 207 lines of code, and 91.73s for zero knowledge control flow analysis on the Solovay-strassen with 301 lines of code using SNARKs. The proof size is only 128 bytes and the verification time is 1.4ms. Using Spartan improves the prover time by 6.6–10.8×, with an increase on the proof size and the verification time.

For the same analysis, the performance largely depends on the size of the program and the maximum number of variables in a function. In addition, the interval analysis is more expensive than the other two. This is because the transfer function, compare and join operations for the interval analysis contains many comparisons on the bounds of intervals, while those for the tainting and control flow analysis can be efficiently implemented as a small circuit. Moreover, the worklist algorithm for the tainting and control flow analysis converges very fast and the number of iterations is only slightly larger than the number of flows. On the contrary, in the interval analysis the intervals are updated many times when seeing loops. Thanks to our widening technique with a threshold of 5, the number of iterations is around 5 times of the number of flows.

The performance is reasonable in practice to prove properties of secret programs with soundness and zero-knowledge. One can also take advantages of other zero-knowledge proof backends to achieve better trade-offs on the performance for different applications.

6.2 Comparing to Generic Schemes

We then compare the performance of our zkAI scheme with RAM-based generic zero knowledge proof systems [16, 66]. The scheme in [16] proposes a universal RAM-to-circuit reduction supporting arbitrary RAM programs written in an assembly-like language named TinyRAM. Each cycle costs around 4000 gates. We implement the worklist algorithm in TinyRAM, count the number of cycles and estimate the size of the circuit. The scheme in [66], referred as Buffet, automatically compiles a subset of C language to a program-specific circuit. Though the original scheme in [66] uses a ZKP backend with circuit-dependent trusted setup, one can replace it by backends without trusted setup to construct zero knowledge abstract interpretation schemes per Definition 2. Therefore, here we only compare with the frontend of Buffet. We implement the worklist algorithm in the syntax of Buffet and compile it using the open-source implementation [7].

Figure 4 shows the size of the circuits produced by the three schemes for the tainting analysis on synthetic program with 10 variables. As shown in the figure, the general-purpose RAM-to-circuit-reduction introduces the highest overhead per instruction/-cycle. The size of the circuits is 605 – 6400× larger than our scheme and is beyond the current scale of the ZKP backend. The compiler of Buffet significantly improves the size of the circuits by utilizing program-dependent optimizations. However, the circuits are still 35 – 40× larger than our schemes manually constructed for the tainting analysis. Compiling the worklist algorithm and the abstract interpretation in general to arithmetic circuits automatically and efficiently is an interesting future research, and we believe the techniques proposed in this paper could potentially help the design.

6.3 Evaluations on Synthetic Programs

Finally, we further demonstrate the scalability of our schemes. We follow the pattern of the real programs, and generate synthetic programs of different sizes and parameters for benchmarking.

For the tainting analysis, our synthetic program contains nested function calls. The tainting sources hide in some functions and the

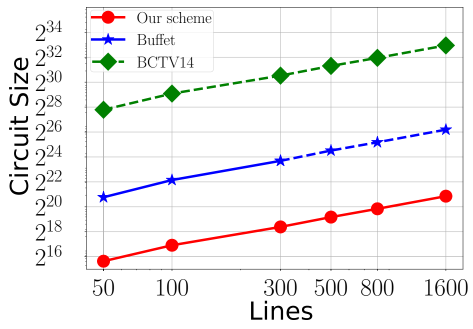


Figure 4: Comparison to generic zero-knowledge proofs

	Tainting	Interval	CFA
#lines	12,800	2,000	4,000
#variables	70	50	50
#other parameters	/	40% in loop	Multiplicity=8
#iterations	25,231	5,075	4,317
circuit size	47.8 M	41.3 M	57.0 M
prover time [61]	406 s	394 s	421 s
verification time	65.8 s	57.9 s	75.7 s
proof size	282KB	282 KB	282KB

Table 3: Performance of our zkAI schemes on large synthetic programs with Spartan [61] as the backend.

tainting information is supposed to be passed to the caller in nested function calls. The functions are calling each other randomly, ensuring that each function is called at least once. We set the maximum number of variables in a function as 70. With these parameters, we then randomly generate the statements inside each function. For the interval analysis, our synthetic program contains nested loops with memory accesses. We set some array buffers to have limited sizes, and access the buffer in instructions later. The buffer overflow error can then be detected in the problematic nested loops. To deal with loops with many iterations, we take a widening strategy that whenever the interval of a variable is updated more than a certain number of times, we expand it to $(-\infty, \infty)$. In our experiments, we set this threshold to 5. We set the maximum number of variables in a function as 50. In addition, we also set the percentage of the code contained in the loops as 40%. For the control flow analysis, we set up some identity functions at the beginning, and use some branches in the main function to adjust possible functions a variable can refer to. We define 'multiplicity' as the maximum number of functions a variable can refer to. We set multiplicity as 8, and the maximum number of variables in a function as 50. We then generate the remaining program using random statements.

We plot the size of the circuits produced by our zero-knowledge abstract interpretation schemes for the three analyses on synthetic programs with different parameters and sizes in Figure 6 in Appendix E. We scale all programs to the largest instances that can be handled on our machine, and we show the performance of the largest programs in Table 3. As shown in the table, we are able to perform the tainting analysis on a program with 12,800 lines of code, the interval analysis with 2,000 lines of code and the control flow analysis with 4,000 lines of code. The R1CS produced by our zkAI is 41.3 to 57 million constraints. We are able to run the backend of Spartan [61] on these R1CS instances. The prover time for the tainting analysis for example is 406s, the proof size is 282KB and the verifier time is 65.8s. The SNARK backend [42] runs out of memory, but can actually scale to half of these largest instances. The prover time is estimated to be around 4500s, with a proof size of 128 bytes and verification time of 1.4ms.

Experimental Results for Proving Absence of Bugs. We pick two examples from the experiments: the tainting analysis on synthetic programs with 50 variables and the interval analysis on synthetic programs with 50 variables and 20% of code in the loops. We implement their corresponding validation algorithms in Algorithm 5 and Figure 5 shows their circuit sizes. As shown in the

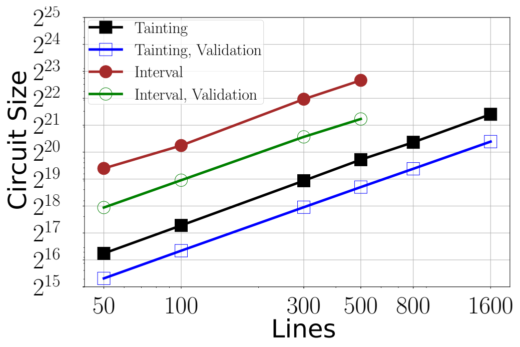


Figure 5: Performance of zkAI for proving absence of bugs.

figure, the circuit size is 1.9-2× smaller in the tainting checks, and 2.4-2.8× smaller in the interval analysis. The main reason is that the validation algorithm only goes through all the flows once, while the worklist algorithm iterates till convergence. The improvement is roughly proportional to the ratio of the number of iterations over the number of flows. The savings on the push operations in the queue and the join operations in the lattice turn out to be small.

7 CONCLUSION

We have demonstrated an application of zero knowledge proofs to static analysis of programs, and specifically using the framework of abstract interpretation. We describe both intra-procedural and inter-procedural analyses for a core imperative language, implementation details for efficient execution in zero knowledge, and evaluation results showing that the approach is practical in real settings.

Although we focus on the scenario where a prover wishes to demonstrate knowledge that a secret program is free of bugs, there are broader applications of zero knowledge abstract interpretation to explore in future work, for example, in accelerating the performance of demonstrating proof-of-exploit in zero knowledge. Approaches for proof-of-exploit have been explored in prior work, and are based in simulating the concrete execution of a program from a particular input which reaches the exploit. In principle, applying the abstract interpretation framework to compute sound *under*-approximations—as opposed to sound *over*-approximations, the focus in our work—could accelerate the efficiency of zero knowledge proof of exploit by orders of magnitude. Such an approach could be checked efficiently by the verifier without computing fixpoints (as we show for over-approximations in Section 5) and provide an irrefutable guarantee that the exploit exists (i.e., no false positives).

ACKNOWLEDGMENT

We thank Michael Walfish for helpful discussions which inspired this work. This material is based upon work supported by DARPA under Contract No. HR001120C0087 and by NSF under awards CCF-2119939. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

REFERENCES

[1] [n.d.]. Buffet. https://github.com/pepper-project/releases.

[2] [n.d.]. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa.

[3] [n.d.]. Children's Online Privacy Protection Rule ("COPPA"). https://www.ftc.gov/enforcement/rules/rulemaking-regulatory-reform-proceedings/childrens-online-privacy-protection-rule.

[4] [n.d.]. DroidBench 2.0. https://github.com/secure-software-engineering/DroidBench. Accessed: 2020-12-03.

[5] [n.d.]. General Data Protection Regulation (GDPR) - Official Legal Text. https://gdpr-info.eu/.

[6] [n.d.]. Health Insurance Portability and Accountability Act of 1996. https://www.hhs.gov/hipaa/index.html.

[7] [n.d.]. Pequin: An end-to-end toolchain for verifiable computation, SNARKs, and probabilistic proofs. https://github.com/pepper-project/pequin. Accessed: 2020-12-03.

[8] [n.d.]. Reachability. https://github.com/ilyasergey/reachability.

[9] [n.d.]. WCET Project Benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html. Accessed: 2020-12-03.

[10] 2014. libsnark: a C++ library for zkSNARK proofs. https://github.com/scipr-lab/libsnark.

[11] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. 2017. Ligero: Lightweight sublinear arguments without a trusted setup. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.

[12] Stephanie Bayer and Jens Groth. 2012. Efficient zero-knowledge argument for correctness of a shuffle. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 263–280.

[13] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable zero knowledge with no trusted setup. In *Annual International Cryptology Conference*. Springer, 701–732.

[14] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. [n.d.]. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *CRYPTO 2013*.

[15] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. 2019. Aurora: Transparent Succinct Arguments for R1CS. In *Advances in Cryptology – EUROCRYPT 2019*. Springer International Publishing, 103–128. https://doi.org/10.1007/978-3-030-17653-2_4

[16] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. [n.d.]. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *Proceedings of the USENIX Security Symposium, 2014*.

[17] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Scalable zero knowledge via cycles of elliptic curves. In *CRYPTO 2014*. 276–294.

[18] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. 1994. Checking the correctness of memories. *Algorithmica* 12, 2-3 (1994), 225–244.

[19] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. 2016. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *International Conference on the Theory and Applications of Cryptographic Techniques*.

[20] Jonathan Bootle, Andrea Cerulli, Essam Ghadafi, Jens Groth, Mohammad Hajiabadi, and Sune K Jakobsen. 2017. Linear-time zero-knowledge proofs for arithmetic circuit satisfiability. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 336–365.

[21] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. 2018. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 595–626.

[22] Benjamin Braun, Ariel J. Feldman, Zuocheng Ren, Srinath T. V. Setty, Andrew J. Blumberg, and Michael Walfish. [n.d.]. Verifying computations with state. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP, 2013*.

[23] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. [n.d.]. Bulletproofs: Short Proofs for Confidential Transactions and More. In *Proceedings of the Symposium on Security and Privacy (SP), 2018*, Vol. 00. 319–338.

[24] Matteo Campanelli, Dario Fiore, and Anaïs Querol. [n.d.]. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs.. In *CCS 2019*.

[25] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 738–768.

[26] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2006. Terminator: Beyond Safety. In *Computer Aided Verification*, Thomas Ball and Robert B. Jones (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 415–418.

[27] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. [n.d.]. Geppetto: Versatile Verifiable Computation. In *S&P 2015*.

[28] P. Cousot. 1999. The Calculational Design of a Generic Abstract Interpreter. In *Calculational System Design*, M. Broy and R. Steinbrüggen (Eds.). NATO ASI Series F. IOS Press, Amsterdam.

[29] P. Cousot and R. Cousot. 1976. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*. Dunod, Paris, France, 106–130.

[30] P. Cousot and R. Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, Los Angeles, California, 238–252.

[31] P. Cousot and R. Cousot. 1979. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, San Antonio, Texas, 269–282.

[32] P. Cousot and R. Cousot. 1992. Inductive Definitions, Semantics and Abstract Interpretation. In *Conference Record of the Ninthteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, Albuquerque, New Mexico, 83–94.

[33] Patrick Cousot and Radhia Cousot. 2014. A Galois connection calculus for abstract interpretation. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 3–4.

[34] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. 2005. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming (ESOP'05) (Lecture Notes in Computer Science, Vol. 3444)*, M. Sagiv (Ed.). © Springer, Edinburgh, Scotland, 21–30.

[35] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

[36] Christian Ferdinand and Reinhold Heckmann. 2004. aiT: Worst-Case Execution Time Prediction by Static Program Analysis. In *Building the Information Society*, Renè Jacquart (Ed.). Springer US, Boston, MA, 377–383.

[37] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. 2016. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*.

[38] Robert W Floyd. 1993. Assigning meanings to programs. In *Program Verification*. Springer, 65–81.

[39] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The knowledge complexity of interactive proof systems. *SIAM Journal on computing* 18, 1 (1989), 186–208.

[40] Eric Goubault and Sylvie Putot. 2013. Robustness Analysis of Finite Precision Implementations. In *Programming Languages and Systems*, Chung-chieh Shan (Ed.). Springer International Publishing, Cham, 50–57.

[41] Jens Groth. 2009. Linear algebra with sub-linear zero-knowledge arguments. In *Advances in Cryptology-CRYPTO 2009*. Springer, 192–208.

[42] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*. 305–326.

[43] David Heath and Vladimir Kolesnikov. 2020. Stacked garbling for disjunctive zero-knowledge proofs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 569–598.

[44] Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

[45] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. 2005. Taming False Alarms from a Domain-Unaware c Analyzer by a Bayesian Statistical Post Analysis. In *Proceedings of the 12th International Conference on Static Analysis* (London, UK) *(SAS'05)*. Springer-Verlag, Berlin, Heidelberg, 203–217. https://doi.org/10.1007/11547662_15

[46] John B Kam and Jeffrey D Ullman. 1976. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)* 23, 1 (1976), 158–171.

[47] Ken Kennedy. 1979. *A survey of data flow analysis techniques*. IBM Thomas J. Watson Research Division.

[48] Gary A Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 194–206.

[49] Joe Kilian. 1992. A Note on Efficient Zero-Knowledge Proofs and Arguments (Extended Abstract). In *Proceedings of the ACM Symposium on Theory of Computing*.

[50] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[51] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

[52] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*.

[53] Mary Maller, Sean Bowe, Markulf Kohlweiss, and Sarah Meiklejohn. 2019. Sonic: Zero-knowledge SNARKs from linear-size universal and updatable structured reference strings. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2111–2128.
[54] Silvio Micali. 2000. Computationally Sound Proofs. *SIAM J. Comput.* (2000).
[55] Jan Midtgaard and Thomas Jensen. 2008. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *Static Analysis*, María Alpuente and Germán Vidal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 347–362.
[56] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 2010. *Principles of Program Analysis*. Springer Publishing Company, Incorporated.
[57] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *S&P 2013*. 238–252.
[58] David Pichardie. 2005. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. Ph.D. Dissertation. Université Rennes 1. In french.
[59] Polyspace Code Prover. 2014. Static Analysis with Polyspace Products. *Mathworks, June* (2014).
[60] Jacob T Schwartz. 1979. Probabilistic algorithms for verification of polynomial identities. In *International Symposium on Symbolic and Algebraic Manipulation*. Springer, 200–215.
[61] Srinath Setty. 2020. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *Annual International Cryptology Conference*. Springer, 704–737.
[62] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. 2018. Proving the correct execution of concurrent services in zero-knowledge. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 339–356.
[63] Olin Grigsby Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages of Taming Lambda*. Ph.D. Dissertation. USA. UMI Order No. GAX91-26964.
[64] Julien Signoles, Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, and Boris Yakobowski. 2012. Frama-c: a Software Analysis Perspective. *Formal Aspects of Computing* 27. https://doi.org/10.1007/s00165-014-0326-7
[65] Arnaud Venet and Guillaume Brat. 2004. Precise and Efficient Static Array Bound Checking for Large Embedded C Programs. *SIGPLAN Not.* 39, 6 (June 2004), 231–242. https://doi.org/10.1145/996893.996869
[66] Riad S. Wahby, Srinath T. V. Setty, Zuocheng Ren, Andrew J. Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*.
[67] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 926–943.
[68] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged Abstract Interpreters. (2019).
[69] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation. In *Advances in Cryptology (CRYPTO)*.
[70] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. [n.d.]. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *S&P 2020*.
[71] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *Security and Privacy (SP), 2017 IEEE Symposium on*. IEEE, 863–880.
[72] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. A Zero-Knowledge Version of vSQL. Cryptology ePrint.
[73] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2018. vRAM: Faster verifiable RAM with program-independent preprocessing. In *Proceeding of IEEE Symposium on Security and Privacy (S&P)*.
[74] Richard Zippel. 1979. Probabilistic algorithms for sparse polynomials. In *International Symposium on Symbolic and Algebraic Manipulation*. Springer, 216–226.

A ZERO-KNOWLEDGE ARGUMENTS

A zero-knowledge argument is a protocol between a computationally-bounded prover $\mathcal{P}$ and a verifier $\mathcal{V}$ for an NP relationship $\mathcal{R}$. At the end of the protocol, $\mathcal{P}$ convinces $\mathcal{V}$ that she knows a witness w such that $(x; w) \in \mathcal{R}$ for some input x. "PPT" standards for probabilistic polynomial time. We use $\mathcal{G}$ to represent the algorithm to generate the public parameters. Formally, a zero-knowledge argument of knowledge is defined below, where $\mathcal{R}$ is known to $\mathcal{P}$ and $\mathcal{V}$.

Definition 1. *Let $\mathcal{R}$ be an NP relation. A tuple of algorithm $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a zero-knowledge argument of knowledge for $\mathcal{R}$ if the following holds.*

- ***Completeness**. For every* pp *output by $\mathcal{G}(1^\lambda)$, $(x; w) \in \mathcal{R}$ and $\pi \leftarrow \mathcal{P}(x, w, \mathrm{pp})$, $\Pr[\mathcal{V}(x, \pi, \mathrm{pp}) = 1] = 1$*
- ***Knowledge Soundness**. For any PPT prover $\mathcal{P}^*$, there exists a PPT extractor $\mathcal{E}$ such that given the access to the entire executing process and the randomness of $\mathcal{P}^*$, $\mathcal{E}$ can extract a witness w such that $\mathrm{pp} \leftarrow \mathcal{G}(1^\lambda)$, $\pi^* \leftarrow \mathcal{P}^*(x, \mathrm{pp})$ and $w \leftarrow \mathcal{E}^{\mathcal{P}^*}(\mathrm{pp}, x, \pi^*)$: $\Pr[(x; w) \notin \mathcal{R} \wedge \mathcal{V}(x, \pi^*, \mathrm{pp}) = 1] \leq \mathrm{negl}(\lambda)$.*
- ***Zero-knowledge**. There exists a PPT simulator $\mathcal{S}$ such that for any PPT algorithm $\mathcal{V}^*$, $(x; w) \in \mathcal{R}$,* pp *output by $\mathcal{G}(1^\lambda)$, it holds that $\mathrm{View}(\mathcal{V}^*(\mathrm{pp}, x)) \approx \mathcal{S}^{\mathcal{V}^*}(x)$.*

*We say that $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ is a **succinct** argument system if the total communication between $\mathcal{P}$ and $\mathcal{V}$ (proof size) is $\mathrm{poly}(\lambda, |x|, \log |w|)$.*

In the definition of zero-knowledge, $\mathrm{View}(\mathcal{V}^*(\mathrm{pp}, x))$ denotes the veiw the verifier sees during the execution of the interactive process with $\mathcal{P}$ while $\mathcal{S}^{\mathcal{V}^*}(x)$ denotes the view generated by $\mathcal{S}$ given input x and transcript of $\mathcal{V}^*$, and $\approx$ denotes two distributions perfect indistinguishable. This definition is commonly used in existing transparent zero-knowledge proof schemes [11, 15, 23, 67].

In addition, in order to build our zero-knowledge abstract interpretation scheme, we need an additional property formalized as "Commit-and-Prove" in [24]. It allows the prover to commit to the witness first, and later prove statements about the committed value. It is naturally supported by most of ZKP systems. We denote the algorithm as $\mathrm{com}_w \leftarrow \mathrm{Commit}(w, \mathrm{pp})$. It is executed after $\mathcal{G}$ and before $\mathcal{P}$, and $\mathcal{V}$ additionally takes com_w as an input. It satisfies the extractability of commitment. Similar to the extractability in Definition 1, there exists a PPT extractor $\mathcal{E}$, given any tuple $(\mathrm{pp}, x, \mathrm{com}^*_w)$ and the executing process of $\mathcal{P}^*$, it could always extract a witness w^* such that $\mathrm{com}^*_w \leftarrow \mathrm{Commit}(w^*, \mathrm{pp})$ except for negligible probability in λ. Formally speaking, $\mathrm{com}^*_w = \mathrm{Commit}(\mathcal{E}^{\mathcal{P}^*}(\mathrm{pp}, x, \mathrm{com}^*_w), \mathrm{pp})$.

Protocol 1 (Zero-Knowledge Abstract Interpretation(zkAI)). *Let λ be the security parameter, $\mathbb{F}$ be a prime field, p be the secret program, C be the arithmetic circuit in Figure 2. Let $\mathcal{P}$ and $\mathcal{V}$ be the prover and the verifier respectively. We use $\mathrm{ZKP}.\mathcal{G}$, ZKP.Commit, $\mathrm{ZKP}.\mathcal{P}$, $\mathrm{ZKP}.\mathcal{V}$ to represent the algorithms of the backend ZKP protocol.*

- $\mathrm{pp} \leftarrow \mathrm{zkAI}.\mathcal{G}(1^\lambda)$: $\mathrm{pp} = \mathrm{ZKP}.\mathcal{G}(1^\lambda)$
- $\mathrm{com}_p \leftarrow \mathrm{zkAI.Commit}(p, \mathrm{pp})$: $\mathrm{com}_p = \mathrm{ZKP.Commit}(p, \mathrm{pp})$.
- $\pi \leftarrow \mathrm{zkAI}.\mathcal{P}(p, (L', f', g, \mathrm{Alg}), \mathrm{pp})$:
  (1) *$\mathcal{P}$ runs the algorithm* Alg *with input p, L' and f' to get $S = \mathrm{Alg}(p, L', f')$. Then generates the witness $w = (CFG, Tr)$ for the circuit C during the procedure of the abstract interpretation algorithm. CFG and Tr represents the extended witness in Figure 2. Let $\mathrm{com}_w \leftarrow \mathrm{ZKP.Commit}(w, \mathrm{pp})$. $\mathcal{P}$ sends com_w to $\mathcal{V}$.*
  (2) *After receiving the randomness $\mathbf{r}'$ for checking consistency of the program and the control flow graph, $\mathcal{P}$ invokes $\mathrm{ZKP}.\mathcal{P}(C, p, \mathbf{r}', w, pp)$ to get π. Sends π to $\mathcal{V}$.*
- $\{0, 1\} \leftarrow \mathrm{zkAI}.\mathcal{V}(\mathrm{com}_p, \mathrm{com}_w, (L', f', \mathrm{Alg}, g), \pi, \mathrm{pp})$: *$\mathcal{V}$ outputs 1 if $\mathrm{ZKP}.\mathcal{V}(C, \mathrm{com}_p, \mathbf{r}', \pi, \mathrm{com}_w, pp) = 1$, otherwise it outputs 0.*

B DEFINITIONS OF ZERO-KNOWLEDGE ABSTRACT INTERPRETATION

Definition 2. *We say that a scheme is a zero-knowledge abstract interpretation if the following holds:*

- ***Completeness.*** *For any program p and analysis $(val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g)$, $\mathsf{pp} \leftarrow \mathsf{zkAI}.\mathcal{G}(1^\lambda)$, $\mathsf{com}_p \leftarrow \mathsf{zkAI.Commit}(p, \mathsf{pp})$, $(y, \pi) \leftarrow \mathsf{zkAI}.\mathcal{P}(p, (val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), \mathsf{pp})$, it holds that*

$$\Pr\left[\mathsf{zkAI}.\mathcal{V}(\mathsf{com}_p, (val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), y, \pi, \mathsf{pp}) = 1\right] = 1$$

- ***Soundness.*** *For any PPT adversary Adv, the following probability is negligible in λ:*

$$\Pr\left[\begin{array}{l}\mathsf{pp} \leftarrow \mathsf{zkAI}.\mathcal{G}(1^\lambda)\\ (p^*, \mathsf{com}_{p^*}, (val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), y, \pi^*) \leftarrow \mathsf{Adv}(1^\lambda, \mathsf{pp})\\ \mathsf{com}_{p^*} = \mathsf{zkAI.Commit}(p^*, \mathsf{pp})\\ \mathsf{zkAI}.\mathcal{V}(\mathsf{com}_{p^*}, (val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), y, \pi^*, \mathsf{pp}) = 1\\ g(\mathsf{Alg}(p, val^\sharp, \mathcal{A}_{p,l})) \neq y\end{array}\right]$$

- ***Zero-Knowledge.*** *For security parameter λ, $\mathsf{pp} \leftarrow \mathsf{zkAI}.\mathcal{G}(1^\lambda)$, for a program p, PPT algorithm Adv, and simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$, consider the following two experiments:*

```
$\mathsf{Real}_{\mathsf{Adv},p}(\mathsf{pp})$:
  (1) $\mathsf{com}_p \leftarrow \mathsf{zkAI.Commit}\ (p, \mathsf{pp})$
  (2) $(val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g) \leftarrow \mathsf{Adv}(\mathsf{com}_p, \mathsf{pp})$
  (3) $(y, \pi) \leftarrow \mathsf{zkAI}.\mathcal{P}(p, (val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), \mathsf{pp})$
  (4) $b \leftarrow \mathsf{Adv}(\mathsf{com}_p, (val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), y, \pi, \mathsf{pp})$
  (5) Output $b$
```

```
$\mathsf{Ideal}_{\mathsf{Adv},\mathcal{S}^{\mathsf{Adv}}}(\mathsf{pp}, h)$:
  (1) $\mathsf{com} \leftarrow \mathcal{S}_1(1^\lambda, \mathsf{pp})$
  (2) $(val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g) \leftarrow \mathsf{Adv}(\mathsf{com}, \mathsf{pp})$
  (3) $(y, \pi) \leftarrow \mathcal{S}_2^{\mathsf{Adv}}(\mathsf{com}, (val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), \mathsf{pp})$,
  (4) $b \leftarrow \mathsf{Adv}(\mathsf{com}, (val^\sharp, \mathcal{A}_{p,l}, \mathsf{Alg}, g), y, \pi, \mathsf{pp})$
  (5) Output $b$
```

For any PPT algorithm Adv and all programs p, there exists simulator $\mathcal{S}$ such that the following probability is $\mathsf{negl}(\lambda)$:

$$|\Pr[\mathsf{Real}_{\mathsf{Adv},p}(\mathsf{pp}) = 1] - \Pr[\mathsf{Ideal}_{\mathsf{Adv},\mathcal{S}^{\mathsf{Adv}}}(\mathsf{pp}, h) = 1]|.$$

C PROOF OF MEMORY CHECKING

Theorem 2. *Let M_0 be the initial state of the memory of size m known to the verifier. Let $(a_1, a_2, ..., a_t)$ be the sequence of addresses to access. Following the procedure mentioned in Section 3.2 to construct RS and WS set, with the additional requirement that $t \leq ts$ at each step ,if the prover manages to give M_t that has the same size as M_0, and satisfy $M_0 \cup WS = M_t \cup RS$, then all the values in RS given by the prover is consistent with values in the memory computed honestly.*

Proof. We use RS_i, WS_i to denote the first i element of RS and WS respectively, i.e., the read set and write set after step i. We use M_i to denote the content of memory after step i.

First, observe that if the prover faithfully computes the sets and the memory up to step i, then $M_i = M_0 \cup WS_i \setminus RS_i$. We would like to prove that if the prover gives a wrong value pair in the sequence of memory accesses, then it is impossible for him to give a final M_t such that $M_0 \cup WS = M_t \cup RS$.

Now let us consider the first inconsistent value in the sequence of memory accesses, happening at step j, where $1 \leq j \leq t$ w.l.o.g. Since the prover faithfully gives the correct value up to step $j-1$, we can write $M_{j-1} = M_0 \cup WS_{j-1} \setminus RS_{j-1}$, which is the state of the memory before step j. If the prover gives the value pair $(a_j, (v'_{a_j}, t'_{a_j}))$ that is inconsistent with the real value $(a_j, (v_{a_j}, t_{a_j}))$, then this means it is not in M_{j-1}. Moerover, this fake value pair can not appear in following $WS \setminus WS_{j-1}$ because all value pairs in $WS \setminus WS_{j-1}$ has a time step larger than t'_{a_j} which is ensured by the condition $t \leq ts$ at each step, and ts is increasing by 1. As a result, it is impossible to find M_t such that equation 1 holds

$$M_{j-1} \cup (WS \setminus WS_{j-1}) = (RS \setminus RS_{j-1}) \cup M_t, \quad (1)$$

because the fake pair is not in M_{j-1} or $(WS \setminus WS_{j-1})$, but in $(RS \setminus RS_{j-1})$. Finally, recall that $M_{j-1} = M_0 \cup WS_{j-1} \setminus RS_{j-1}$. Substituting it into Equation 1 shows that $M_0 \cup WS = RS \cup M_t$ cannot hold. □

D PROOF OF THEOREM 1

Completeness. As explained in Section 3.2, the circuit in $\mathsf{zkAI}.\mathcal{P}$ outputs 1 if $g(\mathsf{Alg}(p, L', f')) = 1$. Therefore, the correctness of Protocol 1 follows the zero-knowledge proof protocol by Theorem 2.

Soundness. By the extractability of the commitment in the zero knowledge proof backend (Definition 1), with overwhelming probability, there exists a PPT extractor $\mathcal{E}$ such that given com_w, it extracts a witness w^* such that $\mathsf{com}_w = \mathsf{ZKP.Commit}(w^*, \mathsf{pp})$. By the soundness of zkAI in Definition 2, if $\mathsf{com}_p = \mathsf{zkAI.Commit}(p, \mathsf{pp})$ and $\mathsf{zkAI}.\mathcal{V}(\mathsf{com}_p, (L', f', \mathsf{Alg}, g), \pi, \mathsf{pp}) = 1$ but $g(\mathsf{Alg}(L', f')) = 0$, let $\mathsf{com}_w = \mathsf{ZKP.Commit}(w^*, \mathsf{pp}_2)$ during the interactive process in Protocol 1, then there are two cases.

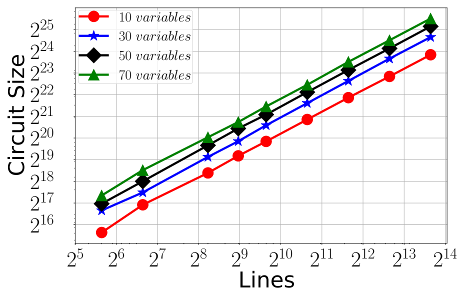
- Case 1: $w^* = (CFG^*, Tr^*, r)$ such that $C((\mathsf{com}_p, CFG^*, Tr^*, \mathbf{r}'); w^*) = 1$. Then we know either the control flow graph is not consistent with the program representation or the iteration check fails. The probability of both events are $\mathsf{negl}(\lambda)$ as claimed by the soundness of the checks in 3.2. Hence, the probability that $\mathcal{P}$ could generate such w^* is also $\mathsf{negl}(\lambda)$ by the union bound.
- Case 2: $w^* = (CFG^*, Tr^*, r)$ but $C((\mathsf{com}_p, CFG^*, Tr^*, \mathbf{r}'); w^*) = 0$. Then according to the soundness of Aurora, given the commitment com^*_w, the adversary could generate a proof π_w making $\mathcal{V}$ accept the incorrect witness and output 1 with probability $\mathsf{negl}(\lambda)$.

Combining these two cases, the soundness of the zkAI scheme is also $\mathsf{negl}(\lambda)$.

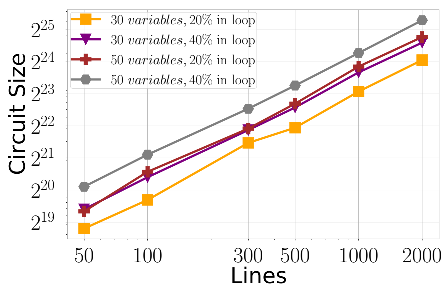
Zero-knowledge. The zero-knowledge property follows directly from the commitment scheme and the zero-knowledge backend we use.

E ADDITIONAL EXPERIMENTAL RESULTS

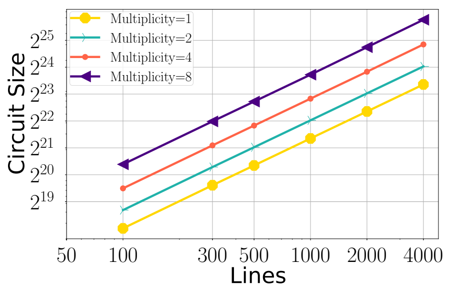
Circuit size for our zkAI on synthetic programs. The size of circuits produced by our zero-knowledge abstract interpretation schemes for various analyses and sizes are shown in Figure 6.



(a) Tainting Analysis



(b) Interval Analysis



(c) Control Flow Analysis

Figure 6: Circuit size of our zkAI schemes.