Ou: Automating the Parallelization of Zero-Knowledge Protocols

Yuyang Sang[*]
Yale University
New Haven, USA
yuyang.sang@yale.edu

Ning Luo[*]
Northwestern University
Evanston, USA
ning.luo@northwestern.edu

Samuel Judson
Yale University
New Haven, USA
samuel.judson@yale.edu

Ben Chaimberg
Yale University
New Haven, USA
ben.chaimberg@yale.edu

Timos Antonopoulos
Yale University
New Haven, USA
timos.antonopoulos@yale.edu

Xiao Wang
Northwestern University
Evanston, USA
wangxiao@northwestern.com

Ruzica Piskac
Yale University
New Haven, USA
ruzica.piskac@yale.edu

Zhong Shao
Yale University
New Haven, USA
zhong.shao@yale.edu

ABSTRACT

A zero-knowledge proof (ZKP) is a powerful cryptographic primitive used in many decentralized or privacy-focused applications. However, the high overhead of ZKPs can restrict their practical applicability. We design a programming language, Ou, aimed at easing the programmer's burden when writing efficient ZKPs, and a compiler framework, Lian, that automates the analysis and distribution of statements to a computing cluster. Lian uses programming language semantics, formal methods, and combinatorial optimization to automatically partition an Ou program into efficiently sized chunks for parallel ZK-proving and/or verification. We contribute:

(1) A front-end language where users can write proof statements as imperative programs in a familiar syntax;
(2) A compiler architecture and implementation that automatically analyzes the program and compiles it into an optimized IR that can be lifted to a variety of ZKP constructions; and
(3) A cutting algorithm, based on Pseudo-Boolean optimization and Integer Linear Programming, that reorders instructions and then partitions the program into efficiently sized chunks for parallel evaluation and efficient state reconciliation.

CCS CONCEPTS

• **Security and privacy → Formal methods and theory of security**.

KEYWORDS

Programming language, Zero-knowledge proofs, Parallelization

ACM Reference Format:
Yuyang Sang[*], Ning Luo[*], Samuel Judson, Ben Chaimberg, Timos Antonopoulos, Xiao Wang, Ruzica Piskac, and Zhong Shao. 2023. Ou: Automating the Parallelization of Zero-Knowledge Protocols. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3616621

CCS '23, November 26–30, 2023, Copenhagen, Denmark

ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3616621

1 INTRODUCTION

Zero-knowledge proofs (ZKPs) [24] enable a prover to convince a verifier about the truth of some statement without revealing why. Recent advances by the cryptographic research community have brought a tremendous improvement in the efficiency of general-purpose zero-knowledge (ZK) proving (see [1] for a comprehensive survey), as well as numerous creative applications of ZKPs (e.g., [3, 6, 10, 12, 21]). However, various practical challenges make the adoption of ZK unfeasible for tasks of realistic sizes. A particular problem is the lack of a comprehensive toolkit of programming languages and compilers providing a full suite of effective, intuitive, *parallel-aware*, and general methods for writing complex statements intended for efficient ZKPs.

Challenge 1: Simple and scalable development. One significant challenge for applying ZKPs to new problems is describing the statement to be proven in a way that is at the same time natural to the programmer and conducive to compilation.

Most existing languages and compilers for ZK only support the 'circuit model' based on *data-oblivious* computation, where control flow must be independent of all private variables (e.g., Snarky, Cairo). They encode the statement as a monolithic program and emit an appropriate circuit representation for their target protocol. Although this model is theoretically complete [25, 32], it does not capture the full flexibility and power of modern ZKP constructions. For example, many statements can be accelerated by nondeterminism (*e.g.*, [7, 8], and recent language support [9]), where the prover provides extra hints to the computation. Other statements (*e.g.*, 3-coloring and Hamiltonian cycles in graphs [23]) can be verified with high concrete efficiency under a probabilistic guarantee: the verifier provides random challenges, to which the prover replies with challenge-dependent hints.

We contribute a more general-purpose ZK programming framework able to support proofs over multiple computational models. Our aim is to ease development for new domains – especially so for

[*] These authors contributed equally to this work.

ZK statements that are more naturally expressed in those alternative models, or for which those models provide superior performance.

Challenge 2: Scalability in hardware resources. All sufficiently advanced technology may be indistinguishable from magic [18], but the 'magic' of ZK often depends as much in practice on massive hardware resources as on mathematical ingenuity. These demands lead to practical bottlenecks in computation and/or communication resources when scaling ZKPs to large statements. For example, most zkSNARK protocols [11, 14, 26, 35, 38] require temporary storage linear in the running time of the computation being proved. Although VOLE-based ZKPs [4, 20, 40] use less memory, they only do so at the expense of a significant increase in required bandwidth.

To aid practicality, a line of recent work has started to "scale out" ZK protocols: instead of assuming the prover and/or verifier have access to one giant machine, they assume access to a cluster of machines and the capability to distribute the task amongst these resources. For example DIZK [41] distributes the prover computation of Groth's ZK protocol [27] by manually partitioning the R1CS constraints into equal-sized chunks. zkBridge [42] can distribute the prover computation of Virgo [44] efficiently to multiple servers when the statement already has a high degree of parallelism. Meanwhile, EZEE [43] distributes among multiple machines the prover-verifier computation and communication of an interactive, garbled-circuit-based zero-knowledge proof protocol [30]. Giraffe [37] addresses the challenge of automatically distributing computations in the verifiable outsourcing context by breaking down overly large computations that cannot be outsourced directly.

Though efficient in deployment, in order to distribute their chosen ZK protocol over a cluster of servers each of these works require the developer to first both i) modify the ZKP protocols to be amenable to distributed computation; and ii) to manually partition the computation into smaller chunks to be proven in parallel. Such manual processes may be error-prone and lead to suboptimal solutions, especially for statements of real-world complexity.

1.1 Summary of Contributions

In this paper, we design a programming framework – a language Ou and a supporting compiler architecture Lian – enabling programmers to write zero-knowledge statements without thinking about distributed computation. Our compiler automatically and efficiently chunks this program for efficient parallel proving.

(1) Our framework provides a C-like programming language with annotations so that developers, including those without deep knowledge of cryptography, can easily write ZK applications even when using protocols with advanced features.

(2) We develop a compiler and static analysis pipeline, written in OCaml, that calculates the local and communication costs of each program instruction, then automatically finds an efficient way to chunk the corresponding ZK statement. The optimality measure minimizes the sum of the maximum per-machine cost and cumulative communication costs. This analysis can distribute ZK statements incorporating randomized verification.

(3) We prove our system sound, *i.e.*, we show the semantics of the distributed verification of substatements is equivalent to verifying the whole statement centrally.

```
#define S 100
#define secparam 128
plocal1 plc1 int[S][S] mmult_plcmx(...) {...}
pvt2 int[S] mmult_pvtvec(...) {...}
pvt2 int[S] mmult_pubvec(...) {...}

void frvlds(pvt1 int [S][S] M,
            pvt1 int [S][S] M1, pvt1 int [S][S] M2) {
    /* array of verifier controlled randomness */
    pub2 int [S] s = {0};

    int t = 0;
    /* repeat for secparam number of times */
    while (t < secparam) {
        int i = 0;
        while (i < S) {s[i] = v_rand(0, 1); i = i+1;}

        /* compute z = (M1 * (M2 * s)) */
        pvt2 int[S] w = mmult_pubvec(M2, s);
        pvt2 int[S] z = mmult_pvtvec(M1, w);

        /* compute q = (M * s) */
        pvt2 int[S] q = mmult_pubvec(M, s);

        /* check that q and z are the same */
        i = 0;
        while (i < S) {
            assert (q[i] == z[i]);  i = i+1;
        }
        t = t+1;
    }
}

unit mmult(plc1 int [S][S] M1, plc1 int [S][S] M2) {
    /* prover locally computes M1 * M2 = M */
    plc1 int[S][S] M = mmult_plcmx(M1, M2);
    return frvlds(commit(M), commit(M1), commit(M2));
}
```

Figure 1: Running example. We consider this Ou implementation of Freivalds' algorithm for randomized verification of 100x100 matrix multiplication.

(4) Empirically, we show that for many statements, including gradient descent and Merkle Tree, our framework can automatically find efficient partitioning of the statements, thus achieving speedup proportional to the number of machines available.

What this paper is not about. Lian does not establish a program's functional correctness or data-obliviousness, nor the hardness of witness finding for the ZK statement. We assume a program written in our language is sound and secure for its intended ZK application, and is suitably constructed for the target protocol: *e.g.*, we assume the program is data-oblivious if the target is circuit-based.

1.2 Technical Challenges and Solutions

Ou and Lian address a series of challenges arising from each of language design, cryptography, and combinatorial optimization.

Cryptographic insight. To enable arbitrary partitioning of a statement, the underlying ZK protocol needs to be 'flexible': it must allow proving multiple statements of the same witness while ensuring consistency. To this end, our framework focuses on the commit-and-prove paradigm. The prover first commits to their private inputs, and then constructs a proof establishing some (public) relationship among the committed values without revealing them to the verifier. Because the inputs are committed, the prover can establish multiple statements on them, all while the verifier is certain their values remain consistent. Any ZKP system can support this capability by proving the consistency of commitments in ZK, but certain protocols have direct support which incurs less cost (*e.g.*, [2, 4, 15, 16, 19, 29, 30, 40]). Given such a ZK backend secure under parallel composition, regardless of its internals, e.g., the use of Fiat-Shamir or random oracle, we can then distribute the proof of any ZK statement in parallel.

During compilation, we first partition the statement into multiple substatements, before the prover precomputes the input/output of each substatement locally. The prover can then in parallel prove to the verifier the correctness of each substatement and then establish the consistency of the input/output between substatements. Although this works in principle, designing a scalable system that automatically distributes the computation with minimum runtime- and developer-overhead is still challenging. For example, in ZK data dependency rarely constrains statement partitioning, unlike in normal computation or MPC where it does so always. So in ZK even inherently sequential statements can often be parallelized.

Automatic statement partitioning. In ZK, the prover knows both the public and private inputs and can thus 'predict' all intermediate values. As a result, data dependencies can be resolved before proving: computation of $y = f(g(x_1), x_2)$ can be partitioned into two parallel verification tasks, $t = g(x_1)$ and $y = f(t, x_2)$. The prover, with inputs x_1 and x_2, just appends t to form an *extended witness*. For ordinary computation, this parallelization would be infeasible because f depends on g. The prover's ability to 'locally precompute' creates new opportunities to maximize parallelism.

To partition effectively, after parsing and typechecking, our architecture undertakes two distinct special compilation phases: *shallow simulation* and *deep simulation*. The first automatically partitions the ZK statement and then selects variables as witnesses (the appendices to the input, like t) for each of the κ substatements. The parameter κ is user-chosen, and will equal the number of available compute cores in the cluster. Deep simulation then uses the prover's private inputs to concretely compute the values of those extended witnesses, as well as of any public variables.

Shallow simulation proceeds by using live variable analysis to compute the costs of each instruction in the statement, and then models them as a directed acyclic graph for partitioning by either *pseudo-Boolean optimization* (PBO) or *integer linear programming* (ILP). This requires loop unrolling and function inlining as flattening techniques. The resultant ZK substatements are then compiled to the chosen backend protocol implementation for distribution as programs to the participating servers: in our evaluation, we use the VOLE-based EMP-toolkit [39]. Lian's deep simulation then propagates input-dependent values through the statement to concretely compute both public values and the prover's witnesses for each substatement. To distinguish which values are available during our different simulation phases, we use *knowledge levels*: K_0 for (private input-independent) values that are known at compile time, and K_1 for (private input-dependent) values that are known at distribution time. This way, reusing a proof with different inputs only requires recomputing the deep simulation, and not the shallow simulation or partitioning, where the latter is likely to be the most computationally intensive compilation task.

Supporting randomized verification. As noted, randomized verification can be much more efficient than deterministic verification. However, supporting it during shallow simulation in particular is a challenge. With deterministic verification, an oblivious program trace is known at compile time, which allows Lian to analyze the cost of each substatement easily. With randomized verification, the program might branch on the at-the-time unknown randomness, preventing the cost from being calculated precisely as whether an expensive statement is entered may depend on a coin flip. Also, privacy imparts an ordering requirement on randomized verification: the verifier's randomness used in checking a ZK statement can only be revealed after the prover commits to its inputs. This means that the partitioning needs to obey sequential ordering for values that are dependent on dynamic choices of the verifier.

Ou uses a third knowledge level, K_2, to indicate that the value of a variable can only be known at runtime. The type system enforces non-interference with respect to a security lattice, allowing K_0- and K_1-leveled variables to flow into K_2-leveled variables, but not the other way around. The partitioning then treats the K_2-leveled variables as one contiguous block in order to estimate worst-case cost and enforce sequential ordering.

Handling large statements. Naively implementing the aforementioned techniques does not scale to large ZK statements. If the cost graph contains too many nodes, finding efficient chunks will be computationally infeasible. Classical graph partitioning and many related variants are NP-hard with efficient (approximate) optimization algorithms known only in certain special cases, and even then often only theoretically [13]. More subtly, a naive representation of each substatement, *e.g.*, as a circuit or R1CS, is necessarily linear in its running time. For large statements, especially those containing unrolled loops or inlined function calls, it is therefore crucial to work with 'compact representations' of substatements that are sublinear in the running time.

Ou and Lian support sublinear representations through `atomic`, a user-provided annotation to indicate that a function should be condensed, rather than flattened, during shallow simulation. By doing so, we contract the component of the cost graph corresponding to the function into a single node of cost cumulative of its constituent instructions. This reduces the size of the graph and makes optimization more practical. Even if the `atomic` function is called multiple times, only one copy of the declaration exists. This brings the partitioning problem into the realm of practicality for both PBO and ILP solving, despite its inherent hardness.

1.3 Roadmap

Sec. 2 is an overview of the Lian framework's workflow. Sec. 3 presents the language's syntax, typing rules, and dynamic semantics. Sec. 4 shows how the shallow simulation unfolds the program into a sequential program, and formally proves the two programs have the same behavior. Sec. 5 then describes how the compiler finds an efficient way to partition the sequential program into multiple chunks and generate a distributed program. Sec. 6 shows how the deep simulation evaluates all dependent data between chunks. It also proves that with these data, the distributed program has the same behavior as if it was run without distribution. Finally, Sec. 7 studies the effectiveness of the framework for distributing the computations of various benchmarks.

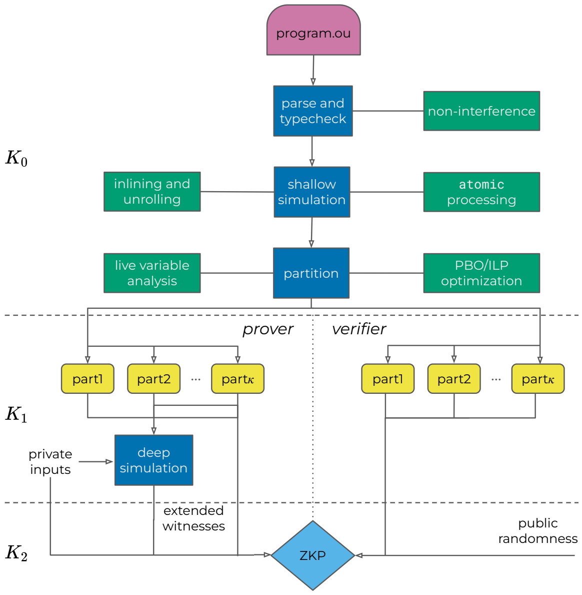


Figure 2: The Lian framework, beginning with an Ou program and ending in binaries for an implementation of a ZKP protocol. Square blocks with light text identify the compiler components: the operations (blue) and their underlying methods (green). Dark text indicates a program or protocol: the Ou program (purple semi-rounded rectangle), compiled substatement binaries (gold rounded rectangles), and the ZKP protocol itself (light blue diamond). Corresponding knowledge levels are displayed as well.

2 LIAN DESIGN OVERVIEW

Figure 2 shows the architecture and workflow of the Lian framework. The user first writes a program in Ou. Upon invocation, the compiler parses and typechecks the program. The Ou type system enforces non-interference over the security lattice given in Figure 3. This lattice specifies non-interference along two dimensions:

i the public or private nature of the information; and
ii the knowledge level of the information (i.e., if the information is available at compile time, distribution time or run time)

Accordingly, the lattice defines seven security domains: public `pub`$\times \{K_0, K_1, K_2\}$, committed and authenticated `pvt` $\times \{K_1, K_2\}$, and prover-local `plc` $\times \{K_1, K_2\}$. As syntactic sugar, we write `pub0` for (`pub`, K_0) and similarly for the rest of the lattice. The variables in the public domain are known by both the prover and the verifier, the variables in the prover-local domain are only known by the prover, and the variables in the authenticated domain are only known by the prover but the verifier holds a commitment to them. Since values known at compile time can only be public, (`pvt`, K_0) and (`plc`, K_0) do not exist.

The authenticated and prover-local variables contain information computed from the prover's secret inputs. Public variables are usually used to guide the program's execution, either deterministically as iterators or fixed constants, or as verifier-provided randomness. In a valid Ou program, every variable is annotated by the security label of a position in the lattice.[1] For example, in Figure 1 `pub0` annotates the public matrix size constant `S` that is known at compile time. In comparison, further up the lattice `pvt2` = (`pvt`, K_2) annotates `w`, which depends on both the prover's committed private values (`M2`, necessitating `pvt`) and on the verifier's public randomness (`s`, necessitating K_2).

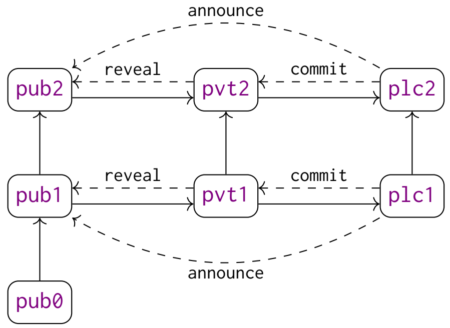


Figure 3: Security Lattice

The security lattice enforces that the private and prover-local domains' information never flow into the public domain except by explicit lowering through a `reveal` operation. This non-interference property [22, 33] allows the safe partial execution of the program during compile time (the shallow simulation), as the control flow can be pre-determined. The inclusion of the knowledge levels in the security lattice is essential to guaranteeing that `reveal` operations and public verifier-randomness are safely handled under the non-interference guarantee.

Once typechecking completes shallow simulation commences. Shallow simulation yields a sequential program consisting of a flattened sequence of operations through loop unrolling and function inlining. With two exceptions, every instruction in the original Ou program forms its own independent block in this sequential program: the constituent instructions of functions decorated by the `atomic` keyword are combined into a single block, as are all K_2-annotated variables. In both cases, this mapping enforces them to be treated as one cumulative 'instruction' for the purposes of partitioning. In Figure 1 for example, the suffix of the program beginning with the first `mmult_pubvec` call on L19 will be condensed, thereby guaranteeing the security of the parallel composition in the presence of public randomness.

If the user sets $\kappa = 1$, so that only a single partition is desired, then this sequential program will be directly compiled into the ZK statement as specified by the backend. However, usually $\kappa > 1$ as the developer desires parallelization, and we generally assume this to hold through the remainder of our discussion. Next, the Lian compiler conducts live variable analysis to track dependencies among the blocks and builds a corresponding directed acyclic graph $G = (V, E)$. An edge $(i, j) \in E$ captures that j-th instruction block depends on some information computed in the i-th instruction block. For example, in Figure 1 the `commit(M)` on L37 depends on the `mmult_plcmx` block on L36.

In order to obtain efficient paritions, the results of the live variable analysis are used to decorate G with both edge and node labels, each indicating a form of cost to the optimality of any partition. Nodes in G are labeled with their *computation cost*, which models the time required by the prover and verifier to execute the block's proving or verification operations in the chosen ZKP protocol backend. Edges are labeled with the *cutting cost*, which models the communication cost required to reconcile the state of the blocks

[1] The parser accepts unannotated declarations, which are interpreted as `pub0`.

across compute nodes in the cluster. The latter is linear to the operation's inbound dependent data. We assume that cutting and computation costs are directly comparable through some scaling factor (α), but otherwise omit that detail as it is irrelevant to the compiler's function.

Lian next uses combinatorial optimization to find the efficient-cuts to partition G. This optimization problem is most easily understood as an ILP, though as the constituent variables are all Booleans, it is also amenable to representation as a PBO problem.

We formally give the opt$(\cdot)$ function for this problem, as well as all of its constraints, in Sec. 5.2. Informally, the objective can be understood as the maximum of the cumulative computation costs of each partition, plus the sum of all cutting costs on edges bisected by the $\kappa - 1$ cuts (needed to create the user-specified κ chunks). Users can adjust the parameters of the ILP/PBO instances or objective functions to align with different ZK backends and relevant metrics.

The partitioned program is compiled into binaries for the chosen ZKP protocol backend. As Lian is designed to use commit-and-prove ZK protocols as a black box without any internal changes to them, our highly adaptable framework can support a wide range of backends, including the NIZK protocols obtained via the Fiat-Shamir transform [5]. These binaries are then sent to all individual provers and verifiers. All computation up to this point is based only on public K_0 information, and so the binaries may be freely distributed and reused. However, for the programs to run in parallel, the dependent values between partitions must be computed ahead of time. This is the deep simulation. The prover uses their private information to simulate execution of each substatement program, and saves the output values of each one that then become the inputs to others. These output values are the 'extended witnesses' of Figure 2. For example, in Figure 1 a cut at L12 would make each entry in `M` an extended witness. After deep simulation completes, the prover can, with any verifier, execute the ZKP protocol by running their binaries in parallel and connecting each binary to the other party's running copy over the network (*i.e.*, part0 to part0, part1 to part1, and so on).

Finally, after all the chunks have been checked in parallel, their inputs and outputs must be reconciled to check consistency of variables passed across the cuts. This step is essential, otherwise the prover could violate soundness by inconsistently computing their extended witnesses, so that, *e.g.*, some variable $t = 0$ in one partition but $t = 1$ in another. Therefore the runtime of the proving is not only a function of how large each of the partitioned substatements are, but also of how much data must be communicated amongst the compute cores during this consistency checking step – justifying its inclusion in the optimization objective. Lian automatically compiles this reconciliation step into the binaries themselves, so it requires no additional expertise on behalf of the verifier.

3 OU LANGUAGE

3.1 Language Syntax

Figure 4 demonstrates Ou's syntax. Note that we use a^+ as shorthand for one or more non-terminals separated by commas like $a_1, \cdots, a_n$, and use a^* to represent zero or more non-terminals likewise. A program consists of a collection of struct declaration (sd), function declaration (fd), and external function declaration (fx) statements. There must exist a `main` function acting as the entry point to the entire program. The syntax is very similar to many imperative languages, except for the addition of *security levels* ℓ to label primitive types and *annotations* η to label certain functions.

$$
\begin{array}{rcl}
\ell & ::= & \texttt{pub0} \mid \texttt{pub1} \mid \texttt{pub2} \mid \texttt{pvt1} \mid \texttt{pvt2} \mid \texttt{plc1} \mid \texttt{plc2} \\
\tau & ::= & \ell\ \textbf{int} \mid \ell\ \textbf{bool} \mid \ell\ \textbf{float} \mid \textbf{unit} \mid \tau\ [\ell\ \texttt{num}] \mid \textbf{struct}\ s \mid \tau\ * \\
 & & \\
e & ::= & \ell\ \texttt{num} \mid \ell\ \texttt{true} \mid \ell\ \texttt{false} \mid \texttt{null} \mid \texttt{tt} \mid (e) \\
 & & \mid \mathfrak{f}(e^*) \mid \ell\ \{e^+\} \mid \{(.\texttt{label} = e)^+\} \mid \epsilon \mid \&\ \epsilon \\
\epsilon & ::= & x \mid \epsilon[e] \mid \epsilon.\texttt{label} \mid *\ e \\
c & ::= & \tau\ x = e \mid \epsilon = e \\
 & & \mid \tau\ x = f(e^*) \mid c_1; c_2 \mid \{c\} \mid \textbf{assert}\ e \\
 & & \mid \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{while}\ e\ \textbf{do}\ c \\
 & & \mid \textbf{return}\ e \mid \textbf{return} \mid \textbf{break} \mid \textbf{continue} \\
 & & \\
\eta & ::= & \texttt{normal} \mid \texttt{atomic} \mid \texttt{box1} \mid \texttt{box2} \mid \texttt{plocal1} \mid \texttt{plocal2} \\
sd & ::= & \textbf{struct}\ s = \{(\tau\ \texttt{label})^+\} \\
fd & ::= & \eta\ \tau\ f((\tau\ x)^*)\{c\} \\
fx & ::= & \eta\ \tau\ f((\tau)^*) \\
prog & ::= & (sd|fd|fx)^*
\end{array}
$$

Figure 4: Syntax of the Ou language

Each atomic type τ is annotated with a security level ℓ. Arrays have both a τ indicating their elements' type, as well as an ℓ indicating their *access level.* The access level can be viewed as the security level of indexes. A struct's type only mentions its name, and not its fields, so that we can support recursive struct definitions.

There are two different kinds of expressions: L-expressions (ϵ) and R-expressions (e). An L-expression refers to a memory location like a variable x, an array index $\epsilon[e]$, a struct field $\epsilon.l$, or a dereferenced pointer $* e$. An R-expression represents a value like an integer, float, or Boolean constant; the calling of a builtin function, array constructor, or struct constructor; or the loading of an L-expression or a pointer to an L-expression. When constructing an array $\ell\ \{e_0, \ldots, e_{n-1}\}$, the user has to specify its access level. The user can also write normal binary and unary operators, including typecasting following the arrows in Figure 3, and all typical arithmetic operations on integers, floats, and Booleans, but Lian will automatically convert them into builtin function calls. For example, given a variable `x` of type `pvt1` **`int`** and a variable `y` of type `pub0` **`int`**, `x + y` will be automatically rewritten into the expression `pvt1_int_add(x, pvt1_int_of_pub0(y))`.

A command c can be a variable declaration, an assignment, the calling of a user-defined function, conditional branching, a `while` loop, a sequence of commands $c_1; c_2$, a block of commands, `return`-ing from a function, or `break`-ing out of a loop.

Note the syntax treats builtin functions and user-defined functions differently. A builtin function $\mathfrak{f}$ can be called in expressions freely, while a user-defined function f has to be received immediately by a variable. This distinction is needed because all builtin functions are pure while user-defined functions can have side-effects. As shallow simulation will reorder computations, having side-effects in expressions can alter behavior. We do still allow calling user-defined functions freely in any expression, but just like in C the programmer cannot assume any execution order as Lian will automatically rewrite the code to extract these calls.

Each function also has an annotation η which enforces restrictions on its body. Most common are **`normal`** functions (the parser interprets unannotated functions as `normal`) which only permit branching on `pub0` Boolean expressions. An `atomic` function is similar, except that shallow simulation will not unfold its body. To branch on a `pub1` value, the user must write the code inside a `box1` function, while `box2` is similar except for additionally allowing branching on `pub2` variables. Functions annotated by `plocal1` or `plocal2` are prover-local functions, which only manipulate `plc1` or `plc2` values respectively. We call functions annotated by `box` or `plocal` *sandboxed* functions, as the type system enforces restrictions to isolate their effects. We further elaborate in Sec. 3.2.

Before typechecking, Lian first scans the program and collects some top-level definitions: Θ maps a struct name s to its definition $\{l_1 : \tau_1, l_2 : \tau_2, \dots\}$. Λ^u maps a user-defined function name f to its definition or declaration. When f is defined, $\Lambda^u(f)$ is like $\textbf{internal}\ (\eta, (\tau_1\ x_1, \dots, \tau_1\ x_n) \rightarrow \tau, c)$ consisting of its annotation, parameter list, return type, and function body; When f is external, $\Lambda^u(f)$ records only $\textbf{external}\ (\eta, (\tau_1, \dots, \tau_1) \rightarrow \tau)$. Note the language does not support global variables, so the prover can only use external functions to communicate her secret with the program. In addition, the language has builtin functions' typing information Λ^b, which map a builtin function name $\mathfrak{f}$ to its signature $(\tau_1, \dots, \tau_n) \rightarrow \tau$. Builtins do not have an explicit function body c, but in Sec. 3.3 we will assign each a math interpretation $(\!|\mathfrak{f}|\!)$.

In order to ensure the security of ZKP programs, it is crucial for users to label witness variables as `pvt` accurately. Failing to do so can lead to information leakage. Lian offers provably-sound typing rules that enable users to implement secure Ou programs. By implementing a typechecker for these rules, Lian can identify and report any errors resulting from incorrect usage of security levels or annotations that violate the typing rules.

3.2 Typing Rules

Figure 5 demonstrates the language's typing rules. Each typing rule relies on the defined struct definitions Θ and function definitions Λ^u and Λ^b. As these are all fixed we omit them from our written rules for concision. We also rely on a typing environment Γ, which is a list of scopes $\gamma_1 \cdots \gamma_n$. Each scope γ_i maps variable names to their types. All newly defined variables reside in the first scope γ_1. Note that though the compiler allows the same name to be defined in different scopes, for clarity our discussion assumes that names never conflict.

Typing orders. We say $\ell_1 \le \ell_2$ if there's a path without a dotted arrow from ℓ_1 to ℓ_2 in Figure 3. We say a type τ is *wellformed* if:

- it is an atomic type like ℓ **int** or ℓ **bool**;
- it is **struct** s such that s is defined in Θ and the types of all its fields are wellformed; or
- it is an array of type $\tau\ [\ell\ n]$ such that τ is wellformed and ℓ is no greater than any security level in τ.

For the last condition, the relevant security levels include those of the atomic type as well as the array's access levels. Wellformedness guarantees the values loaded from arrays are never less secure than the access needed to load them. This restriction is important for both shallow simulation and deep simulation as they need to fully evaluate all K_0 expressions and K_1 expressions, respectively. For example, if $a[e]$ loads a `pub0` integer from array a with `pvt1` access, then shallow simulation does not have enough knowledge to evaluate e, thus it can not know which value is loaded. So although $a[e]$'s type is `pub0`, the shallow simulation cannot evaluate it. To make shallow and deep simulation behave as expected, we assume all types in this paper are wellformed.

Typing expressions. For L-expressions, $\Gamma \vdash_L \epsilon : \tau, \ell$ means in the typing environment Γ an L-expression ϵ points to a value of type τ, and the L-expression itself has access level ℓ. The access level can help us determine whether an L-expression can be fully evaluated during shallow simulation (`pub0`) or deep simulation (`pub1`, `pvt1` and `plc1`) or neither (`pub2`, `pvt2` and `plc2`). Similarly, for R-expressions, $\Gamma \vdash_R e : \tau$ means an R-expression e is of type τ in the typing environment Γ.

Note that in practice the Lian compiler is more forgiving than the language's strict typing rules, as it performs implicit type conversion along the security lattice's arrows. For example, an expression e of type `pub0` **int** can be used as a `pvt1` **int**, and Lian will automatically wrap it inside a casting function `pvt1_int_of_pub0(e)`.

We also require all pointers to have `pub0` access level. This restriction is important to the live variable analysis needed for partitioning. Suppose we were to allow a pointer p to have a `pvt1` access level: then shallow simulation could not evaluate the pointer itself in order to determine which location is mutated after running $* p = e$. This would mean all locations in the stack could potentially be the address. Similarly, loading a value from $* p$ would mean any location could potentially be read. Therefore this restriction on pointer access levels is needed to make the live variable analysis yield meaningful results, rather than just concluding by labeling all variables as live.

Typing commands. When typechecking a command c which constitutes a function's body, we need its return type τ_r to guarantee that each `return` command in c yields a valid value. We also need a set of branching security levels L_b to guard all branching expressions inside c. In `normal` and `atomic` functions, all branching expressions have to be `pub0`, but prover-local and boxed functions enforce fewer restrictions.

Typing a program. The type checker first checks all struct definitions, and makes sure there are no name collisions between struct names and field names, and then checks each function's body. For a function definition $\eta\ \tau\ f(\tau_1\ x_1, \dots, \tau_n\ x_n)\{c\}$, we apply the command's typing rule on its body: $\Gamma, \tau, L_b \vdash \{c\} : \Gamma$, where $\Gamma = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$ and L_b is determined by η:

$$L_b = \begin{cases} \{\texttt{pub0}\} & \text{if } \eta \text{ is } \texttt{atomic} \text{ or } \texttt{normal} \\ \{\texttt{pub0}, \texttt{pub1}\} & \text{if } \eta \text{ is } \texttt{box1} \\ \{\texttt{pub0}, \texttt{pub1}, \texttt{pub2}\} & \text{if } \eta \text{ is } \texttt{box2} \\ \{\texttt{plc1}\} & \text{if } \eta \text{ is } \texttt{plocal1} \\ \{\texttt{plc1}, \texttt{plc2}\} & \text{if } \eta \text{ is } \texttt{plocal2}. \end{cases}$$

There are additional restrictions placed on sandboxed functions. We require that `plocal1` and `box1` functions only modify external data that is K_1 or K_2, and only return K_1 or K_2 results. Similarly, `plocal2` and `box2` functions can only modify external data that is K_2, and only produce K_2 results. The reason is similar to the restrictions on arrays' access levels: shallow simulation cannot execute `plocal1` or `box1` functions, so it is infeasible to allow them to modify or return any K_0 data. If they could, simulation of any other computations that rely on such data could not proceed after the function call. We also restrict that `plocal` functions can only manipulate `plc`

$$\boxed{\Gamma \vdash_L \epsilon : \tau, \ell}$$

$$\frac{x : \tau \in \gamma}{\gamma \cdot \Gamma \vdash_L x : \tau, \texttt{pub0}} \qquad \frac{\Gamma \vdash_L \epsilon : \tau[\ell\ n], \ell' \qquad \Gamma \vdash_R e : \ell\ \texttt{int} \qquad \ell \leq \ell'}{\Gamma \vdash_L \epsilon[e] : \tau, \ell} \qquad \frac{\Gamma \vdash_L \epsilon : \textbf{struct}\ s, \ell \qquad s : m \in \Theta \qquad l : \tau \in m}{\Gamma \vdash_L \epsilon.l : \tau, \ell} \qquad \frac{\Gamma \vdash_R e : \tau *}{\Gamma \vdash_L * e : \tau, \texttt{pub0}}$$

$$\boxed{\Gamma \vdash_R e : \tau}$$

$$\frac{}{\Gamma \vdash_R \ell\ n : \ell\ \texttt{int}} \qquad \frac{\Gamma \vdash_L \epsilon : \tau, \ell}{\Gamma \vdash_R \epsilon : \tau} \qquad \frac{\Gamma \vdash_L \epsilon : \tau, \texttt{pub0}}{\Gamma \vdash_R \& \epsilon : \tau *} \qquad \frac{\forall i \in [0, n-1],\ \Gamma \vdash_R e_i : \tau}{\Gamma \vdash_R \ell\ \{e_0, \ldots, e_{n-1}\} : \tau[\ell\ n]} \qquad \frac{\mathfrak{f} \mapsto ((\tau_1, \ldots, \tau_n) \to \tau) \in \Lambda^b \qquad \forall i \in [1, n],\ \Gamma \vdash_R e_i : \tau_i}{\Gamma \vdash_R \mathfrak{f}(e_1, \ldots, e_n) : \tau}$$

$$\boxed{\Gamma, \tau_r, L_b \vdash c : \Gamma'}$$

$$\frac{\gamma \cdot \Gamma \vdash_R e : \tau}{\gamma \cdot \Gamma, \tau_r, L_b \vdash \tau\ x = e : \gamma \cup \{x : \tau\} \cdot \Gamma} \qquad \frac{\begin{array}{c} f \mapsto \textbf{internal}\ (\eta, (\tau_1\ x_1, \ldots, \tau_n\ x_n) \to \tau, c) \in \Lambda^u \\ \text{or } f \mapsto \textbf{external}\ (\eta, (\tau_1, \ldots, \tau_n) \to \tau) \in \Lambda^u \\ \forall i \in [1, n],\ \gamma \cdot \Gamma \vdash_R e_i : \tau_i \end{array}}{\gamma \cdot \Gamma, \tau_r, L_b \vdash \tau\ x = f(e_1, \ldots, e_n) : \gamma \cup \{x : \tau\} \cdot \Gamma} \qquad \frac{\begin{array}{cc} \Gamma \vdash_R e : \ell\ \textbf{bool} & \Gamma, \tau_r, L_b \vdash c_1 : \Gamma_1 \\ \ell \in L_b & \Gamma, \tau_r, L_b \vdash c_2 : \Gamma_2 \end{array}}{\Gamma, \tau_r, L_b \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \Gamma}$$

$$\frac{\Gamma \vdash_R e : \tau \qquad \Gamma \vdash_L \epsilon : \tau, \ell}{\Gamma, \tau_r, L_b \vdash \epsilon = e : \Gamma}$$

$$\frac{\Gamma \vdash_R e : \ell\ \textbf{bool}}{\Gamma, \tau_r, L_b \vdash \textbf{assert}\ e : \Gamma} \qquad \frac{\Gamma \vdash_R e : \tau_r}{\Gamma, \tau_r, L_b \vdash \textbf{return}\ e : \Gamma} \qquad \frac{\Gamma, \tau_r, L_b \vdash c_1 : \Gamma_1 \qquad \Gamma_1, \tau_r, L_b \vdash c_2 : \Gamma_2}{\Gamma, \tau_r, L_b \vdash c_1; c_2 : \Gamma_2} \qquad \frac{\Gamma \vdash_R e : \ell\ \textbf{bool} \qquad \ell \in L_b \qquad \Gamma, \tau_r, L_b \vdash c : \Gamma'}{\Gamma, \tau_r, L_b \vdash \textbf{while}\ e\ \textbf{do}\ c : \Gamma}$$

Figure 5: Typing Rules

data. Since all **plocal** functions only run on the prover’s side, they cannot include any pub or pvt values, as their use would require the verifier’s participation.

To enforce the first restriction, we only need to examine the function’s signature: all pointers passed into **plocal1** and **box1** functions should only point to K_1 or K_2 data, while all pointers passed into **plocal2** and **box2** functions should only point to K_2 data. The return types have similar restrictions. Since the language does not support global variables, sandboxed functions can only update external data via the pointers in the arguments. To enforce the second restriction, we need to examine the entire function body to ensure all expressions’ types only have plc annotations and the body only calls other **plocal** functions.

3.3 Dynamic Semantics

This section defines the dynamic semantics that apply if all secrets are known. The semantics correspond to the program’s behavior in the ideal world where it has enough knowledge to fully evaluate all expressions into values and check if all assertions are satisfied. We will use this semantics as a baseline to prove the simulation and distribution are correct. The semantics rules are not surprising. See Appendix B of [34] for detailed definitions.

R-values and L-values. We use v to denote R-values (or “values” for short) and μ to denote L-values. R-values can be atomic values like **Vunit** and **Vint** $\ell\ n$, an empty pointer **Vnull**, a pointer **Vref** μ, an array **Varray** $\ell\ a$ where ℓ is the array’s access level and a is a mapping from indexes to R-values like $\{0 \mapsto v_0, \ldots, n \mapsto v_n\}$, or a struct **Vstruct** m where m is a mapping from label to R-values like $\{l_0 \mapsto v_0, \ldots, l_n \mapsto v_n\}$. An L-value is a reference to a memory location: it can be **Vvar** α that points to an *alias*, or **Vindex** $\mu\ \ell\ i$ that refers to an array μ’s index i, or **Vfield** $\mu\ l$ that refers to a struct μ’s field l. An alias is used by the stack to disambiguate during recursion, and can be viewed as a memory address.

$$\begin{aligned} v ::= {} & \textbf{Vunit} \mid \textbf{Vnull} \mid \textbf{Vref}\ \mu \\ & \mid \textbf{Vint}\ \ell\ \text{num} \mid \textbf{Vfloat}\ \ell\ \text{num} \mid \textbf{Vbool}\ \ell\ \text{bool} \\ & \mid \textbf{Varray}\ \ell\ \text{array} \mid \textbf{Vstruct}\ \text{map} \\ \mu ::= {} & \textbf{Vvar}\ \alpha \mid \textbf{Vindex}\ \mu\ \ell\ \text{num} \mid \textbf{Vfield}\ \mu\ \text{label} \end{aligned}$$

The program runs on a stack Ω, *i.e.*, a list of frames $\omega_1 \cdot \omega_2 \cdots \omega_n$. The last frame ω_n stores main’s variables. The stack grows with nested function calls, and the top frame ω_1 stores the current function call’s local variables. Each frame ω is a map from aliases to values. We use $\Omega[\alpha \mapsto v]$ to denote adding a mapping $\alpha \mapsto v$ to the top frame. Whenever a new variable x is defined, Lian allocates a unique alias $\mathsf{new}(x)$ for it. This is useful when recursion occurs, as the same variable x might have different instances in the stack, so we cannot bind their values to the same name x. We also use a function $\mathtt{find}(\Omega, x)$ to find the latest alias allocated for x in the stack. Note that when x is defined for the first time, $\mathsf{new}(x)$ returns x directly instead of allocating a new alias for it.

Given a stack Ω, we use a function $\mathtt{load}(\Omega, \mu)$ to load the value referred by μ in Ω, and use a function $\mathtt{store}(\Omega, \mu, v)$ to compute the updated stack after replacing the value at μ with v. They both search the entire stack from top frame to bottom instead of working only with the top frame.

Semantics. The semantics looks up the sets of user-defined functions Λ^u and builtin functions Λ^b to interpret function calls. As they are immutable, we omit them from our rules for concision.

Expressions: $[\![\epsilon]\!]_L^\Omega$ computes L-expression ϵ in stack Ω and yields an L-value, $[\![e]\!]_R^\Omega$ evaluates R-expression e and yields an R-value. Note Lian assigns a mathematical interpretation $(\!|\mathfrak{f}|\!)$ for each builtin function $\mathfrak{f}$ which maps a tuple of R-values to an R-value. For example, $(\!|\mathsf{pvt1\_int\_add}|\!)(\textbf{Vint}\ \texttt{pvt1}\ n, \textbf{Vint}\ \texttt{pvt1}\ m) = \textbf{Vint}\ \texttt{pvt1}\ (n+m)$ is assigned by the compiler for integer addition.

Commands: $(c, \Omega_1) \to_C (r, \Omega_2, \beta)$ means that running a command c from a stack Ω_1 yields a sequence of assertion results β, a result r which is either cont or ret v, and a changed stack Ω_2. Since the command constitutes a function’s body, we use ret v to mean the function returns with v and cont to mean it has not yet returned.

4 SHALLOW SIMULATION

Shallow simulation executes and unfolds the program with K_0 knowledge to obtain a simplified sequential program. Lian will attempt as much simplification as possible, and output all unknown computations as a sequence of commands for later analysis and partitioning. All **normal** and **atomic** functions will be executed, but only the former's execution history will be output, while the latter will be output as indivisible calls. As sandboxed functions rely on K_1 or K_2 knowledge, they will not be executed in shallow simulation and will be kept in the generated sequential program.

4.1 Shallow Semantics

This section formalizes the shallow simulation's behavior as a *shallow semantics*. Figure 6 shows its details. The actual implementation uses several optimizations to speed up the simulation.

Symbolic values. To define the semantics of both shallow simulation and deep simulation, we define *symbolic values* to represent R-values not known during the simulation. We do so by redefining the R-value constructor from Sec. 3.3 to support **SVsym** variants, and also use *symbolic stacks*, which store symbolic values.

Builtin functions. During shallow simulation, only a subset of the builtin functions can be interpreted as the rest rely on either K_1 or K_2 knowledge. We use $(\!|\mathfrak{f}|\!)_0$ to denote a shallow simulateable interpretation of a builtin function. It takes in a list of expressions and yields an expression as the result. For example, we have the interpretation $(\!|\texttt{pub0\_int\_add}|\!)_0(\texttt{pub0}\ 1, \texttt{pub0}\ 2) = \texttt{pub0}\ 3$, while $\texttt{pvt1\_int\_add} \notin (\!|-|\!)_0$ as it relies on K_1 knowledge.

Semantics. $[\![\epsilon]\!]^{\Omega}_{L_0} = \epsilon'$ means simplifying an L-expression ϵ in a symbolic stack Ω with K_0 knowledge results in an L-expression ϵ'. The semantics simplifies an L-expression instead of evaluating it directly, as the stack may not contain enough information to fully evaluate it to an L-value. Similarly, $[\![e]\!]^{\Omega}_{R_0} = e'$ simplifies an R-expressions.

When storing a simplified expression into the stack, we use a partial function $\lfloor - \rfloor_{L_0}$ to evaluate an L-expression down to an L-value, and a total function $\lfloor - \rfloor_{R_0}$ to evaluate an R-expression down to a symbolic R-value. $\lfloor - \rfloor_{R_0}$ is able to be total because it replaces all unknown subexpressions by **SVsym**. When loading a value from the stack, we use a total function $\lceil - \rceil_L$ to lift a symbolic L-value to an L-expression, and a total function $\lceil -, - \rceil_R$ to lift a symbolic R-value to an R-expression and concretize any **SVsym** variables using the appropriate L-value. It's formally defined in Appendix C of [34]. We reuse `load` and `store` to load and store symbolic values to and from the symbolic stack.

$$\texttt{sload}(\Omega, \epsilon) = \begin{cases} \lceil \texttt{load}(\Omega, \mu), \mu \rceil_R & \text{when } \lfloor \epsilon \rfloor_{L_0} = \mu \\ \epsilon & \text{otherwise} \end{cases}$$

$$\texttt{sstore}(\Omega, \epsilon, e) = \begin{cases} \texttt{store}(\Omega, \mu, \lfloor e \rfloor_{R_0}) & \text{when } \lfloor \epsilon \rfloor_{L_0} = \mu \\ \Omega & \text{otherwise} \end{cases}$$

When ϵ is not completely known, type wellformedness guarantees that the symbolic value referred to by it must also be unknown, *i.e.*, **SVsym** or an array or struct of unknown values. When so, the value to be stored must also be unknown, thus `sstore` behaves correctly even when ϵ is not fully evaluated.

Commands: $(c, \Omega_1) \rightsquigarrow_C (r, \Omega_2, h)$ means simplifying a command c in a symbolic stack Ω_1 results in: a return status r which is either `cont` or `ret` e, a new stack Ω_2, and a sequence of history h. We use $\cdot$ to denote an empty history.

Shallow simulation does not have enough knowledge to simulate a sandboxed function, thus it returns a symbolic value $\texttt{default}(\tau)$ generated from the type τ. Also, the typing rule guarantees even if a pointer is passed into the function, the pointed data must be filled with **SVsym**. So no matter whether the function has side effect or not, the symbolic stack remains unchanged.

Note that we have added two pseudo-commands **push** and **pop** into the generated program. They help mark the points where a function is called and returned, and their dynamtic semantics are just pushing and popping stacks. Though helpful for the compiler, ignoring them does not change the program's behavior as the alias mechanism guarantees there will never be name conflicts. We introduce them mainly as machinery to help us prove refinement.

Atomic function calls. The size of a generated program can be very large as it is linear in the execution time. To reduce the challenge of partitioning, we allow users to annotate some functions as **`atomic`** to contract the function call into one command in the generated program. Atomic annotations are optional for programs and are intended solely for optimization purposes. The effectiveness of an atomic annotation depends on the user's understanding of the program structure. If an atomic annotation is inappropriate, it may increase the compilation time, but it will *not* compromise the program's security.

Their semantics are very similar to **`normal`** function calls, but instead of generating a sequence of history, an atomic function call $\tau\ x = f(e_1, \ldots, e_n)$ only yields one command in the history: $\tau\ \alpha = f(e'_1, \ldots, e'_n)\ \textbf{R}\ S_R\ \textbf{W}\ S_W$. Here, we add a new command to encode atomic function calls, but its dynamic semantics are the same as for normal function calls. The two extra sets S_R and S_W are only used to help perform live variable analysis.

S_R records a set of L-values defined before calling f that may be read during the function call, while S_W records a set of L-values defined before calling f that may be updated during the function call. They are collected during the simulation using functions from Sec. 5.1: $\texttt{REF}(c)$ computes all L-values that may be read in command c; $\texttt{DEF}(c)$ computes all L-values that may be updated in c.

One convenient fact about shallow simulation is that no pointer dereference will appear in a generated program. It is a corollary of a more essential property: shallow simulation fully evaluates all expressions that only need K_0 knowledge, including all pointers. In Appendix C of [34], we define *normalized* expressions to formally describe this property to help prove the correctness of shallow simulation.

4.2 Correctness of Shallow Simulation

We start by defining a refinement function $\mathcal{R}$ mapping R-values to symbolic R-values. It replaces all non-`pub0` atomic values with **SVsym**. We overload $\mathcal{R}$ to also represent the function that performs the transformations to convert a stack into a symbolic stack.

$$\begin{aligned} \mathcal{R}(\textbf{Vint}\ \ell\ n) &= \begin{cases} \textbf{SVint}\ \ell\ n & \text{when } \ell = \texttt{pub0} \\ \textbf{SVsym} & \text{otherwise} \end{cases} \\ \mathcal{R}(\textbf{Vref}\ \mu) &= \textbf{SVref}\ \mu \\ \mathcal{R}(\textbf{Varray}\ \ell\ a) &= \textbf{SVarray}\ \ell\ \bigcup\{i \mapsto \mathcal{R}(a(i))\} \\ \mathcal{R}(\textbf{Vstruct}\ m) &= \textbf{SVstruct}\ \bigcup\{l_i \mapsto \mathcal{R}(m(l_i))\} \end{aligned}$$

See Figure 7. We also define a semantics $[\![r]\!]^{\Omega}_{\underline{R}}$ in order to evaluate return statuses.

$$[\![\texttt{cont}]\!]^{\Omega}_{\underline{R}} = \texttt{cont} \qquad\qquad [\![\texttt{ret}\ e]\!]^{\Omega}_{\underline{R}} = \texttt{ret}\ [\![e]\!]^{\Omega}_{R}$$

$$
\begin{aligned}
v ::=\ & \mathbf{SVunit} \mid \mathbf{SVnull} \mid \mathbf{SVref}\ \mu \\
\mid\ & \mathbf{SVint}\ \ell\ \mathsf{num} \mid \mathbf{SVbool}\ \ell\ \mathsf{bool} \\
\mid\ & \mathbf{SVfloat}\ \ell\ \mathsf{num} \mid \mathbf{SVsym} \\
\mid\ & \mathbf{SVarray}\ \ell\ \mathsf{array} \mid \mathbf{SVstruct}\ \mathsf{map}
\end{aligned}
\qquad
\begin{aligned}
\llbracket x \rrbracket^{\Omega}_{L_0} &= \mathtt{find}(\Omega, x) \\
\llbracket \epsilon[e] \rrbracket^{\Omega}_{L_0} &= \llbracket \epsilon \rrbracket^{\Omega}_{L_0} [\llbracket e \rrbracket^{\Omega}_{R_0}] \\
\llbracket \epsilon.l \rrbracket^{\Omega}_{L_0} &= \llbracket \epsilon \rrbracket^{\Omega}_{L_0}.l \\
\llbracket * e \rrbracket^{\Omega}_{L_0} &= \epsilon \quad \text{where } \llbracket e \rrbracket^{\Omega}_{R_0} = \&\ \epsilon
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \ell\ n \rrbracket^{\Omega}_{R_0} &= \ell\ n \\
\llbracket \epsilon \rrbracket^{\Omega}_{R_0} &= \mathtt{sload}(\Omega, \llbracket \epsilon \rrbracket^{\Omega}_{L_0}) \\
\llbracket \&\ \epsilon \rrbracket^{\Omega}_{R_0} &= \&\ \llbracket \epsilon \rrbracket^{\Omega}_{L_0} \\
\llbracket \mathfrak{f}(e_1, \ldots, e_n) \rrbracket^{\Omega}_{R_0} &= \begin{cases} (\!|\mathfrak{f}|\!)_0(\llbracket e_1 \rrbracket^{\Omega}_{R_0}, \ldots, \llbracket e_n \rrbracket^{\Omega}_{R_0}) \\ \quad \text{when } \mathfrak{f} \in \mathrm{dom}(\!|-|\!)_0 \\ \mathfrak{f}(\llbracket e_1 \rrbracket^{\Omega}_{R_0}, \ldots, \llbracket e_n \rrbracket^{\Omega}_{R_0}) \quad \text{o/w} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\lceil \mathbf{Vvar}\ \alpha \rceil_L &= \alpha \\
\lceil \mathbf{Vindex}\ \mu\ \ell\ n \rceil_L &= \lceil \mu \rceil_L [\ell\ n] \\
\lceil \mathbf{Vfield}\ \mu\ l \rceil_L &= \lceil \mu \rceil_L.l \\
\lceil \mathbf{SVint}\ \ell\ n, \mu \rceil_R &= \ell\ n \\
\lceil \mathbf{SVref}\ \mu', \mu \rceil_R &= \&\ \lceil \mu' \rceil_L \\
\lceil \mathbf{SVsym}, \mu \rceil_R &= \lceil \mu \rceil_L \\
\lceil \mathbf{SVarray}\ \ell\ a, \mu \rceil_R &= \ell\ \{\lceil a(0), \mathbf{Vindex}\ \mu\ \ell\ 0 \rceil_R, \ldots\} \\
\lceil \mathbf{SVstruct}\ m, \mu \rceil_R &= \{.l_1 = \lceil m(l_1), \mathbf{Vfield}\ \mu\ l_1 \rceil_R, \ldots\}
\end{aligned}
$$

$$
\begin{aligned}
\lfloor \alpha \rfloor_{L_0} &= \mathbf{Vvar}\ \alpha \\
\lfloor \epsilon[\ell\ n] \rfloor_{L_0} &= \mathbf{Vindex}\ \mu\ \ell\ n \text{ when } \lfloor \epsilon \rfloor_{L_0} = \mu \text{ and } \ell = \mathtt{pub0} \\
\lfloor \epsilon.l \rfloor_{L_0} &= \mathbf{Vfield}\ \mu\ l \quad \text{when } \lfloor \epsilon \rfloor_{L_0} = \mu \\
\lfloor * (\&\ \epsilon) \rfloor_{L_0} &= \mu \qquad\qquad\quad \text{when } \lfloor \epsilon \rfloor_{L_0} = \mu \\
\lfloor \ell\ n \rfloor_{R_0} &= \begin{cases} \mathbf{SVint}\ \ell\ n & \text{when } \ell = \mathtt{pub0} \\ \mathbf{SVsym} & \text{o/w} \end{cases} \\
\lfloor \&\ \epsilon \rfloor_{R_0} &= \mathbf{SVref}\ \lfloor \epsilon \rfloor_{L_0} \\
\lfloor \ell\ \{e_0, \ldots\} \rfloor_{R_0} &= \mathbf{SVarray}\ \ell\ \bigcup\{i \mapsto \lfloor e_i \rfloor_{R_0}\} \\
\lfloor \{.l_1 = e_1, \ldots\} \rfloor_{R_0} &= \mathbf{SVstruct}\ \bigcup\{l_i \mapsto \lfloor e_i \rfloor_{R_0}\} \\
\lfloor \epsilon \rfloor_{R_0} &= \mathbf{SVsym} \\
\lfloor f(e_1 \ldots e_n) \rfloor_{R_0} &= \mathbf{SVsym}
\end{aligned}
$$

$$\boxed{(c, \Omega) \rightsquigarrow_C (r, \Omega', h)}$$

$$
\frac{\mathtt{new}(x) = \alpha \qquad \llbracket e \rrbracket^{\Omega}_{R_0} = e'}{(\tau\ x = e, \Omega) \rightsquigarrow_C (\mathtt{cont}, \Omega[\alpha \mapsto \lfloor e' \rfloor_{R_0}], \tau\ \alpha = e')}
\qquad
\frac{\llbracket \epsilon \rrbracket^{\Omega}_{L_0} = \epsilon' \qquad \llbracket e \rrbracket^{\Omega}_{R_0} = e'}{(\epsilon = e, \Omega) \rightsquigarrow_C (\mathtt{cont}, \mathtt{sstore}(\Omega, \epsilon', e'), \epsilon' = e')}
\qquad
\frac{}{(\mathbf{return}\ e, \Omega) \rightsquigarrow_C (\mathtt{ret}\ \llbracket e \rrbracket^{\Omega}_{R_0}, \Omega, \cdot)}
$$

$$
\frac{}{(\mathbf{assert}\ e, \Omega) \rightsquigarrow_C (\mathtt{cont}, \Omega, \mathbf{assert}\ \llbracket e \rrbracket^{\Omega}_{R_0})}
\qquad
\frac{\llbracket e \rrbracket^{\Omega}_{R_0} = \mathtt{pub0}\ \mathtt{true} \qquad (c_1, \Omega) \rightsquigarrow_C (r, \Omega', h)}{(\mathbf{if}\ e\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2, \Omega) \rightsquigarrow_C (r, \Omega', h)}
\qquad
\frac{(c_1, \Omega) \rightsquigarrow_C (\mathtt{cont}, \Omega_1, h_1) \qquad (c_2, \Omega_1) \rightsquigarrow_C (r, \Omega_2, h_2)}{(c_1; c_2, \Omega) \rightsquigarrow_C (r, \Omega_2, h_1; h_2)}
$$

$$
\frac{\begin{gathered} \mathtt{new}(x) = \alpha \quad \eta = \mathtt{normal} \\ f \mapsto \mathbf{internal}\ (\eta, (\tau_1\ x_1, \ldots, \tau_n\ x_n) \to \tau, c) \in \Lambda^u \\ \forall i \in [1, n],\ \llbracket e_i \rrbracket^{\Omega}_{R_0} = e'_i \wedge \mathtt{new}(x_i) = \alpha_i \\ (c, \bigcup\{\alpha_i \mapsto \lfloor e'_i \rfloor_{R_0}\} \cdot \Omega) \rightsquigarrow_C (\mathtt{ret}\ e, \omega \cdot \Omega', h) \\ h' = (\mathbf{push}; \tau_1\ \alpha_1 = e'_1; \ldots; \tau_n\ \alpha_n = e'_n; h; \mathbf{pop}\ \tau\ \alpha = e) \end{gathered}}{(\tau\ x = f(e_1, \ldots, e_n), \Omega) \rightsquigarrow_C (\mathtt{cont}, \Omega'[\alpha \mapsto \lfloor e \rfloor_{R_0}], h')}
\qquad
\frac{\begin{gathered} \mathtt{new}(x) = \alpha \quad \eta \notin \{\mathtt{normal}, \mathtt{atomic}\} \\ f \mapsto \mathbf{internal}\ (\eta, (\tau_1\ x_1, \ldots, \tau_n\ x_n) \to \tau, c) \in \Lambda^u \\ \text{or } f \mapsto \mathbf{external}\ (\eta, (\tau_1, \ldots, \tau_n) \to \tau) \in \Lambda^u \\ \forall i \in [1, n],\ \llbracket e_i \rrbracket^{\Omega}_{R_0} = e'_i \\ \Omega' = \Omega[\alpha \mapsto \mathtt{default}(\tau)] \end{gathered}}{(\tau\ x = f(e_1, \ldots, e_n), \Omega) \rightsquigarrow_C (\mathtt{cont}, \Omega', \tau\ \alpha = f(e'_1, \ldots, e'_n))}
$$

Figure 6: Shallow Semantics

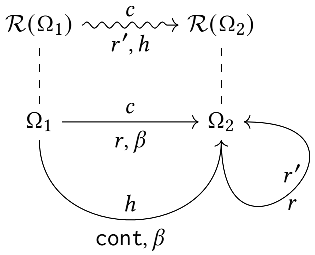


Figure 7: Shallow Semantics Refinement

Theorem 1. *If $(c, \Omega_1) \to_C (r, \Omega_2, \beta)$, then there exist r' and h s.t.*

$$(c, \mathcal{R}(\Omega_1)) \rightsquigarrow_C (r', \mathcal{R}(\Omega_2), h)$$

$$\text{and } (h, \Omega_1) \to_C (\mathtt{cont}, \Omega_2, \beta) \text{ and } \llbracket r' \rrbracket^{\Omega_2}_{\underline{R}} = r.$$

Figure 7 demonstrates this theorem. Although shallow simulation reorders some computations by simplifying K_0 expressions and unfolding the program, this theorem guarantees such reordering does not change the behavior. We prove the theorem in Appendix C.2 of our online full version [34].

5 STATEMENT PARTITIONING

The shallow simulation unfolds the original program into a sequential program $h = c_1; \ldots; c_n$. The compiler then tries to find a way to partition these commands into κ chunks so that the distributed running time and the communication time are minimized. Sec. 5.1 describes the live variable analysis algorithm that finds data dependencies in the sequential program. Sec. 5.2 then describes how Lian encodes the cut search problem and uses a solver to find the efficient partition. Sec. 5.3 then explains how we generate the distributed program from the sequential program, the partition, and the dependencies.

5.1 Live Variable Analysis

Lian performs live variable analysis on a sequential program $h = c_1; \ldots; c_n$, eventually generates a dependency graph $DG = (DV, DE)$. $DV = \{1 \ldots n\}$ and each edge $(i, j) \in DE$ is decorated with a set of L-values $\mathsf{DEP}(i, j)$ that *may* be read in c_j and *may* be manipulated in c_i. We use 'may' rather than 'will' because shallow simulation cannot determine all L-expressions' values. Nonetheless, we can still conduct an over-approximate analysis using some approximation functions. Note $\mathtt{DEP}(i,j)$ is only meaningful when $i < j$, so $\mathtt{DEP}(i,j) = \emptyset$ when $i \geq j$.

Appendix D of [34] defines two utility functions: $\mathtt{REF}(c)$ computes the set of L-values that may be read in c, while $\mathtt{DEF}(c)$ computes the set of L-values that are newly defined or may be updated in c. Recall that we used these functions in Sec. 4.1.

Analysis Algorithm. The program is a sequence of commands $h = c_1; \ldots; c_n$. We start live variable analysis from its end and let $\mathtt{LIVE}(n) := \emptyset$. We then scan from the end to the beginning and compute unresolved dependent variables for each command c_i.

$$\begin{aligned}\mathtt{LIVE}(i-1) = \{\mu \mapsto j \in \mathtt{LIVE}(i) \mid \mu \notin \mathtt{DEF}(c_i) \cup \mathtt{REF}(c_i)\}\\ \cup\, \{\mu \mapsto i \mid \mu \in \mathtt{REF}(c_i)\}\end{aligned}$$

$\mathtt{LIVE}(i)$ is a mapping from L-values to the commands that may read them in the future. In the first line, if c_i uses an L-value μ that is later dependent by c_j,[2] then we can safely delete it from $\mathtt{LIVE}(i)$ and add that L-value as a dependency between c_i and c_j. However, c_i may read other L-values, so in the second line we mark all L-values from $\mathtt{REF}(c_i)$ as unresolved and dependent on c_i. Such dependencies will eventually be resolved because every L-value is part of a variable defined in the program, and the command that defines the variable cannot read it. Now we can construct the graph DG by defining the dependency function:

$$\mathtt{DEP}(i,j) = \{\mu \mid \mu \mapsto j \in \mathtt{LIVE}(i) \wedge \mu \in \mathtt{DEF}(c_i) \cup \mathtt{REF}(c_i)\}$$

In order to capture all dependencies of and from a command, we denote $\mathtt{DEP}(i,-) := \bigcup_{j=1}^{n} \mathtt{DEP}(i,j)$ and $\mathtt{DEP}(-,j) := \bigcup_{i=1}^{n} \mathtt{DEP}(i,j)$. It follows from our definition that DG is a directed acyclic graph, i.e., $i \geq j \implies \mathtt{DEP}(i,j) = \emptyset$. Any L-value dependent on c_i must either be read or written in c_i, any L-value dependent by c_j must be added into the live set during the scan, which can only happen when it is read in c_j, so we have the following lemmas:

Lemma 2. $\mathtt{DEP}(i,-) \subseteq \mathtt{DEF}(c_i) \cup \mathtt{REF}(c_i)$.

Lemma 3. $\mathtt{DEP}(-,j) = \mathtt{REF}(c_j)$.

We need to define a notion of similarity between stacks to reason about the correctness of live variable analysis and program distribution. Given two stacks Ω_1 and Ω_2 and a set of L-values S, we say Ω_1 and Ω_2 are *similar* up to S, denoted as $\Omega_1 \approx_S \Omega_2$, if (1) $\forall \mu \in S, \mathtt{load}(\Omega_1,\mu) = \mathtt{load}(\Omega_2,\mu)$, and (2) the two stacks have similar structures in terms of S, *i.e.*, every alias appearing in S resides in the same frame in both stacks.

Theorem 4 shows the live variable analysis is correct. It says the dependent L-values of c_j are enough to safely execute the command and yield all values that are dependent later. It can be proven using lemma 2 and 3, then prove the values in $\mathtt{REF}(c_j)$ are enough for c_j to execute and produce the desired behavior.

Theorem 4. *If both* $(c_1;\ldots;c_{j-1},\Omega) \to_C (\mathtt{cont},\Omega_{j-1},\beta_{j-1})$ *and* $(c_j,\Omega_{j-1}) \to_C (\mathtt{cont},\Omega_j,\beta)$*, then*

$$\forall \Omega'_{j-1}, \Omega_{j-1} \approx_{\mathtt{DEP}(-,j)} \Omega'_{j-1} \implies$$
$$\exists \Omega'_j, (c_j,\Omega'_{j-1}) \to_C (\mathtt{cont},\Omega'_j,\beta) \wedge \Omega_j \approx_{\mathtt{DEP}(j,-)} \Omega'_j.$$

5.2 Efficient-Cut Search

The final intermediary pass of the compilation partitions the program into κ chunks for parallel execution. Recall that after the flattening during shallow simulation (Sec. 4) and the live variable analysis (Sec. 5.1), Lian obtains a sequential program $h = c_1; \ldots; c_n$ and a dependency graph $DG = (DV, DE)$ where each edge (i,j) in DE is decorated with a set of dependent L-values $\mathtt{DEP}(i,j)$. The compiler then extracts a directed acyclic graph $G = (V,E)$ from DG, capturing the dependency relation and the costs. Each node i in V is decorated with c_i's computation cost PC_i. An edge $(i,j) \in E$ indicates c_j reads at least one L-value used last by c_i, and is decorated with the communication cost C_{ij} and a cuttable bit B_{ij}. If there is a K_2 L-value in $\mathtt{DEP}(i,j)$, then c_i and c_j must reside in the same chunk, thus $C_{ij} = \infty$ and $B_{ij} = 0$; Otherwise, C_{ij} equals to the accumulated cutting cost of all K_1 L-values in $\mathtt{DEP}(i,j)$ and $B_{ij} = 1$.

We use combinatorial optimization constrained by G to find the efficient cutting of h into κ chunks $\langle h_1^{\mathsf{sub}}, h_2^{\mathsf{sub}}, \cdots, h_\kappa^{\mathsf{sub}}\rangle$. We natively write this optimization problem as an integer linear program (ILP). However, the variables (though not the constants) it uses are all Booleans, and therefore the objective can be interpreted as a pseudo-Boolean function and optimized using pseudo-Boolean optimization (PBO) as well.

We begin by defining $ub = \log(\sum_{i=1}^{n} PC_i)$, the length of the cost of the sequential computation. We next define a pair of matrices. $\mathbf{Y} \in \{0,1\}^{n \times n}$, indicates the location of cuts: $\mathbf{Y}_{ij} = 1$ means that a cut is made between the i-th and j-th instruction block. $\mathbf{X} \in \{0,1\}^{n \times \kappa}$ indicates the inclusion of instruction blocks into partitions: $\mathbf{X}_{it} = 1$ indicates the i-th instruction is in the t-th partition for $t \in [\kappa]$.

Finding the cheapest partitioning then reduces to solving the optimization objective of

$$\mathrm{obj}(G) = \min_{\mathbf{X},\mathbf{Y}} \left[\sum_{(i,j) \in E} \mathbf{Y}_{ij} \cdot C_{ij} + \max_{t \in [\kappa]} \sum_{i=1}^{n} \mathbf{X}_{it} \cdot PC_i \right]$$

subject to some consistency constraints on $\mathbf{X}$ and $\mathbf{Y}$. To soundly model this problem for constrained optimization without a min-max objective requires, for PBO in particular, a bit of due care. To convert the original objective into a PBO problem, here we add a bit-vector $\mathbf{COP}$ to encode $\max_{t\in[\kappa]} \sum_{i=1}^{n} \mathbf{X}_{it} \cdot PC_i$. Our approach is closely related to prior work in the literature on PBO for graph partitioning [17]:

$$\begin{array}{lll}
\min & \sum_{i=1}^{ub} 2^{(i-1)} \cdot \mathbf{COP}_i + \sum_{e \in E} C_e \cdot \mathbf{Y}_e & \\
s.t. & \forall i \in [n]: & \sum_{t=1}^{\kappa} \mathbf{X}_{it} = 1 \\
 & \forall (i,j) \in E: & \mathbf{Y}_{ij} \leq B_{ij} \\
 & \forall (i,j) \in E, t \in [\kappa]: & \mathbf{X}_{it} + \mathbf{X}_{jt} \leq 2 - \mathbf{Y}_{ij} \\
 & \forall (i,j) \in E, t \in [\kappa]: & |\mathbf{X}_{it} - \mathbf{X}_{jt}| \leq \mathbf{Y}_{ij} \\
 & \forall t \in [\kappa]: & \sum_{\ell=1}^{ub} 2^{(\ell-1)} \cdot \mathbf{COP}_\ell \geq \sum_{i=1}^{n} \mathbf{X}_{it} \cdot PC_i \\
 & \forall i \in [n], t \in [\kappa]: & \mathbf{X}_{it} \in \{0,1\} \\
 & \forall (i,j) \in E: & \mathbf{Y}_{ij} \in \{0,1\} \\
 & \forall \ell \in [ub]: & \mathbf{COP}_\ell \in \{0,1\}
\end{array}$$

[2] In traditional live variable analysis, a live variable is only resolved when it is modified or defined, but here we also resolve it when it is read. This is a compromise to simplify the efficient-cut search and to reduce cutting cost in most cases. See Appendix D in our online full version [34].

The first constraint enforces that every instruction appears in exactly one partition, the second that only cuttable edges are cut, and the third and fourth together that if the i-th and j-th instruction blocks are in different partitions, then any edge (i, j) must be represented in the appropriate cut by Y_{ij}. The final four constraints simply enforce the Boolean character of the variables.

The fifth constraint is the most nuanced. It enforces that

$$\mathbf{COP}_i = \text{ the } i\text{-th bit of } \max_{t \in [\kappa]} \sum_{i=1}^{n} \mathrm{X}_{it} \cdot PC_i$$

so that the computation in the objective function is correct. This relationship can be encoded much more simply in a general ILP, but is necessary for PBO compatibility. Notice that no constraints force the partitions to be contiguous: even though h is a sequential program, the h_t^{sub} can have a much more complex structure at the cost of additional extended witnesses and so more communication during the consistency checking.

With this encoding, Lian hands this optimization problem to either a PBO or ILP solver (we use the Gurobi PBO solver in our benchmarks). The solver's output may not always be optimal in practice, as the available computational resources limit it. Consequently, the corresponding cut scheme we employ may also be suboptimal. The compiler then converts the result into a function $\mathsf{chunk}(-)$ such that $\mathsf{chunk}(i) = t$ if $X_{it} = 1$ in the solver's result.

Fact 1. $\mathsf{chunk}(i) \neq \mathsf{chunk}(j) \implies$ *no* K_2 *L-value in* $\mathsf{DEP}(i, j)$.

This follows directly from the PBO constraints.

5.3 Distributed Program

Given a sequential program $h = c_1; \ldots; c_n$ from the shallow simulation, a dependency graph $\mathsf{DEP}(-,-)$ from live variable analysis, and a partition $\mathsf{chunk}(-)$ from efficient-cut search, the compiler generates κ programs for distributed execution among κ pairs of prover and verifier computation cores.

The t-th chunk is a sequential program $\mathsf{sync}_{t_1}; c_{t_1}; \mathsf{sync}_{t_2}; c_{t_2}; \ldots$ such that $t = \mathsf{chunk}(t_1) = \mathsf{chunk}(t_2) = \ldots$ and $t_1 < t_2 < \ldots$. We denote a communication between the prover and the verifier by sync_{t_j}. These communication rounds are used to share K_0 or authenticate K_1 data that are needed by c_{t_j}, *i.e.*, data pointed by L-values in $\bigcup_{\mathsf{chunk}(i) \neq t} \mathsf{DEP}(i, t_j)$, as necessary. Deep simulation (Sec. 6) computes all this dependent data and distributes it to all of the prover cores. So each pair of prover core and verifier core can run their chunk without the others' involvement. In the end, all verifier cores will work together to do a consistency check. The goal is to make sure all data provided by the prover cores in sync_{t_j} are indeed the same as produced by their dependent commands in other chunks. Note the deep simulation only computes K_0 and K_1 values, but Fact 1 guarantees all data needed for multipole chunks does not contain K_2 information, thus the deep simulation is enough.

6 DEEP SIMULATION

The deep simulation executes the program with K_0 and K_1 knowledge and evaluates all K_0 and K_1 values. The goal is to compute all data that are dependent between different chunks in the distributed program. It is similar to shallow simulation, except that (i) the definitions of all external functions are known, (ii) the knowledge level is K_1 instead of K_0, and (iii) the purpose is to evaluate all K_0 and all K_1 values instead of unfolding the program.

6.1 Deep Semantics

This section formalizes deep simulation as a *deep semantics*. We use the same symbolic values and stacks as in Sec. 4.1. We still use $\lceil -,- \rceil_R$ and $\lceil - \rceil_L$ to lift R-values and L-values to R-expressions and L-expressions. But when converting in the reversed direction, we use $\lfloor - \rfloor_{R_1}$ and $\lfloor - \rfloor_{L_1}$ instead of $\lfloor - \rfloor_{R_0}$ and $\lfloor - \rfloor_{R_0}$. The only difference is when converting atomic expressions. For example, $\lfloor \ell\ n \rfloor_{R_1} =$ **`SVint`** $\ell\ n$ when $\ell \leq$ `plc1`. We also define `dload` and `dstore` based on these operations just like `sload` and `sstore`.

The semantics for L-expressions $[\![\epsilon]\!]_{L_1}^{\Omega}$ and R-expressions $[\![e]\!]_{R_1}^{\Omega}$ are also similar except for using $\lfloor - \rfloor_{L_1}$ and $\lfloor - \rfloor_{R_1}$ to interact with the symbolic stack, allowing dereference expression to be not fully evaluated, and use $(\!|\mathfrak{f}|\!)_1$ to interpret builtin functions. $(\!|-|\!)_1$ is a superset of $(\!|-|\!)_0$ by adding more functions that can be computed with K_1 knowledge. For example, $(\!|\mathtt{pvt1\_int\_add}|\!)_1(\mathtt{pvt1}\ n, \mathtt{pvt1}\ m) = \mathtt{pvt1}\ (n+m)$. Appendix E in our online full version [34] has their detailed definitions.

Commands' semantics are also similar. $(c, \Omega_1) \hookrightarrow_C (r, \Omega_2)$ means running command c from stack Ω_1 results in a return status r which is either `cont` or `ret` e and a new stack Ω_2. Note it no longer generates a history as the purpose of the deep simulation is merely computing K_0 and K_1 values. Also, it executes **`normal`**, **`atomic`**, **`plocal1`** and **`box1`** functions while ignores **`plocal2`** and **`box2`**. The detailed rules are in Appendix E of [34].

6.2 Correctness of Deep Simulation

Similar to shallow simulation, we define a refinement function $\mathcal{Q}$ that converts R-values to symbolic R-values. This time we only turn all values with `pub2`, `pvt2` and `plc2` security levels into **`SVsym`**. This function is also overridden to apply on stacks.

Theorem 5. *If* $(c, \Omega_1) \rightarrow_C (r, \Omega_2, \beta)$ *then there exists* r' *where*

$$(c, \mathcal{Q}(\Omega_1)) \hookrightarrow_C (r', \mathcal{Q}(\Omega_2)) \text{ and } [\![r']\!]_{\underline{R}}^{\Omega_2} = r.$$

The proof is very similar to theorem 1 as their semantics are similar except for the deep semantics is a bit simpler.

6.3 Correctness of Distribution

Sec. 5.3 shows the distributed program, and Sec. 6.1 evaluates all dependent data that are needed in the `sync` steps. We now proves these data are sufficient for the distributed program to terminate with the same behavior as if running without distribution.

Given a sequential program $c_1; \ldots; c_n$ generated by the shallow simulation and an initial stack Ω_0, suppose the program runs like $(c_i, \Omega_{i-1}) \rightarrow_C (\mathtt{cont}, \Omega_i, \beta_i)$ without cutting. We focus on the t-th chunk and align its execution like in Figure 8. Each command c_j corresponds to two stacks Ω_j^{pre} and Ω_j^{post}. If $\mathsf{chunk}(j) = t$, we first run sync_j to add external dependent values into $\Omega_{j-1}^{\mathsf{post}}$, then run c_j; Otherwise, the stack remains unchanged.

We need to prove that the following theorem holds, then theorem 4 ensures that c_j terminates with the same behavior.

Theorem 6. $\forall j \in \mathsf{chunk}^{-1}(t), \Omega_j^{\mathsf{pre}} \approx_S \Omega_{j-1}$ *where* $S = \mathsf{DEP}(-, j)$.

$$\begin{tikzcd}
\Omega^{\mathsf{post}}_{j-1} \arrow[r, "\mathsf{sync}_j"] & \Omega^{\mathsf{pre}}_{j} \arrow[r, "c_j", "{\mathsf{cont}, \beta}"'] & \Omega^{\mathsf{post}}_{j} \arrow[r, equal] & \Omega^{\mathsf{pre}}_{j+1} \arrow[r, equal] & \Omega^{\mathsf{post}}_{j+1} \\
\Omega_{j-1} \arrow[rr, "c_j", "{\mathsf{cont}, \beta}"'] & & \Omega_j \arrow[rr, "c_{j+1}"] & & \Omega_{j+1}
\end{tikzcd}$$

Figure 8: Execution in a Chunk

In Appendix F of our online full version [34], we prove this theorem by first defining a set of L-values ϕ_j^t that are internally dependent in the t-th chunk after c_j, then prove the stack in the uncut execution is always similar to the t-th chunk's local stack up to this set. So the internal dependency is handled, we then use theorem 5 to handle the external dependency happening in sync, combining them together proves the theorem.

7 EVALUATION

7.1 Evaluation Setup and Metrics

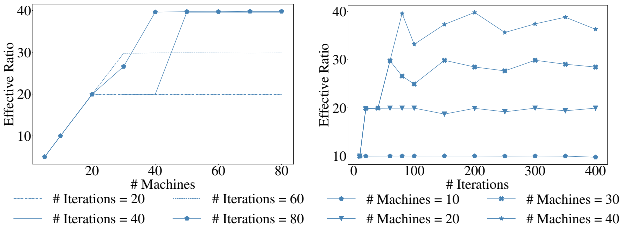
We use three metrics to evaluate the effectiveness of our tool:

- **Compilation Time** is the time used for shallow simulation, data dependency analysis, and efficient-cut search. PBO solving during the compilation is done via the Gurobi [28] PBO solver. We do not include the time cost of deep simulation in the compilation time, but deep simulation takes less than 0.5% of the compilation time for all examples.
- **Effective Ratio** of a distributed zero knowledge program is defined as the quotient of the cost of the sequential execution (total costs of all instructions) by the cost of the distributed execution (as the objective function in Sec. 5). When a program is distributed to κ chunks, the closer the effective ratio is to κ, the better the distribution scheme is. Note that depending on the statement, the optimal effective ratio may be smaller than κ.
- **Execution Time** refers to the end-to-end performance. We compile Ou code to a VOLE-based ZK implementation (EMP [39]) and measure the execution time of the resulting distributed programs. We opt to use VOLE-based ZK backend as one of our motivations is to work with large-scale applications. This choice allows us to handle large statements without requiring costly machines with large memory. In addition, our implementation can be easily extended to support a rich set of ZK protocols since EMP can be connected to SIEVE-IR [36].

To evaluate our tool, we focus on two programs with different features. All implementations in our language Ou can be found in the supplementary material. As the effectiveness of Lian also depends on the structure of the input program, we further provide visualization for the program structure of gradient descent and merkle tree in Appendix G of our online full version [34].

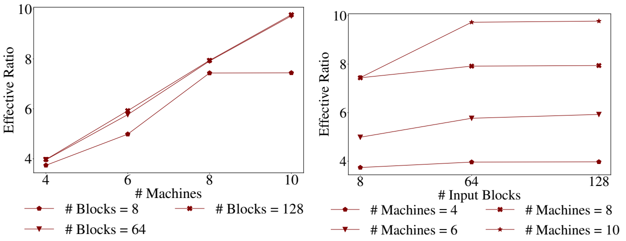
(1) **Gradient descent (GD)** is an optimization algorithm for finding the optimal model in machine learning. It works by iteratively updating the parameters of a model in the direction of the negative gradient of the cost function concerning those parameters. We implement gradient decent for logistic regression with 10 features.

(2) **Merkle tree (MT)** is a type of data structure used to verify the integrity of large amounts of data. It breaks the data into smaller blocks, then recursively hashes each pair of blocks to create a new set of parent nodes using a cryptographic hash function. By proving the correct computation of the Merkle tree, the prover can prove they have the original data without revealing the actual data to the verifier [31].



(a) (b)

Figure 9: The effective ratio for the distribution scheme found by Lian for the GD programs with different numbers of iterations and various numbers of machines (κ). With the increase in κ, the effective ratio will reach the limit. A small-size program with a large κ will make the effective ratio of the distribution scheme much smaller than κ.



(a) (b)

Figure 10: The effective ratio for the distribution scheme found by Lian for the MT programs is close to the numbers of machines (κ) with different numbers of input blocks and κ.

7.2 Evaluation of the Partition Quality

First, we demonstrate the changes in the effective ratio of the distribution scheme found by Lian for GD and MT program changes with the size of programs and the number of cuts. GD takes a set $\{(\mathbf{x}, y)\}$ as input, where $\mathbf{x}$ is a vector of size M. Throughout this section, we fix $|\{(\mathbf{x}, y)\}|$ to be 20 and $|\mathbf{x}|$ to be 10.

The effective ratio of Lian is affected by the length of the program being distributed. The length of GD and MT programs are decided by the number of iterations and the number of input blocks, respectively. As shown in Figure 9b and Figure 10b, Lian can find the distribution scheme that achieves the almost optimal effective ratio for large programs for both GD and MT cases.

Figure 9a and Figure 10a demonstrate that the effective ratio will increase with the number of machines rising when the number of partitions is relatively small. Moreover, the results present an upper bound of the effective ratio of the distribution scheme for a given program. For example, the effective ratio for the GD programs involving 80 iterations reaches 40 and stays unchanged regardless of the increasing number of machines. Nevertheless, the results show that our tool can utilize a high degree of parallelization.

7.3 Effectiveness of Atomic Annotation

In this section, we study the impact of annotation on the compilation time and effective ratio. For all examples, we set the time limit of the PBO solver to 10 minutes. In the GD program, there is a

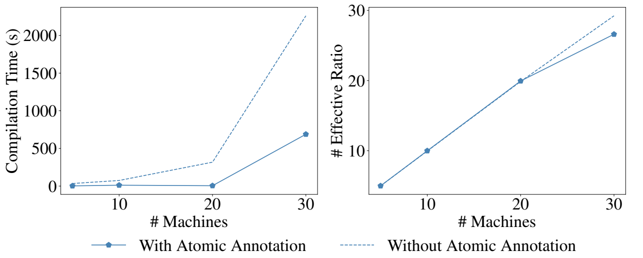


Figure 11: Compiling takes significantly greater time for GD programs when the function `update_all()` is not annotated with the atomic keyword. On the other hand, atomic annotation rules out some feasible solutions, making the distribution scheme slightly suboptimal.

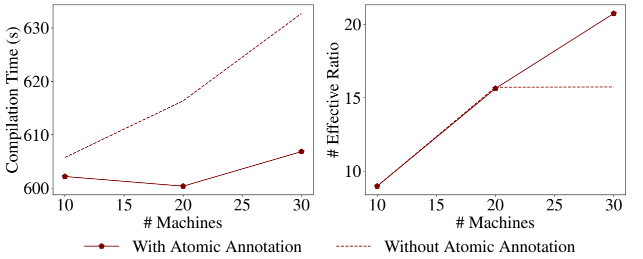


Figure 12: Adding annotation improves both the effectiveness of both compilation and the effective ratio of the MT program for 32-block input.

function called `update_all()`. This function updates the model's parameter using all data points and is called for each iteration. In the MT program, There is a function called `sha256_node()` that computes the hash of internal blocks. We compare the compilation time and effective ratio of GD programs (or MT programs) with and without `update_all()`(or `sha256_node()`) being atomic while changing κ and the number of iterations.

The results of our evaluation show that the effects of adding atomic annotations depend on the structure of the programs. Figure 11 and Figure 12 show a significant rise in the compilation efficiency when adding annotation for both GD and MT programs. The compilation time decreases by 1X to 60X with κ and the number of iterations varying. The price of efficient compilation is a reduction in the effective ratio. The effective ratio for the annotated program is smaller than the unannotated one for GD programs: adding atomic annotation rules out a part of feasible solutions that may contain the optimum.

Sometimes, adding annotation could even improve the effective ratio, especially when the constrained time is far less than what is needed to search for an efficient partition. Figure 12 demonstrates an example of this case. We measure the effective ratio while setting the time limit of the PBO solver to 600 seconds. The effective ratio of the MT program for 32-block input with annotation is even better than the one without annotation when the number of cuts increases. One reason behind this result is the MT program structure. The solution found for the annotated program is also optimum for the unannotated one. Furthermore, with a part of the functions being atomic, the search space of the PBO solving essentially shrinks. For readers who are interested in additional information, we have included the results of MT programs for various input sizes in Appendix G of our online full version [34].

κ	1	5	10	20	40
runtime GD	681.55 s	147.91 s	71.23 s	38.06 s	21.90 s
runtime speedup GD	1	4.6	9.57	17.9	31.12
effective ratio GD	1	4.99	9.95	19.9	39.8
runtime MT	40.23 s	11.08 s	7.63 s	6.5 s	5.25 s
runtime speedup MT	1	3.63	5.27	6.19	7.66
effective ratio MT	1	4.95	9.75	19.47	38.63

Table 1: End-to-end running time of our framework.

7.4 End-to-End Running Time

Our implementation is end-to-end: it can compile an Ou program to κ pieces of EMP circuits. Most of the compiler's implementation is not tied to EMP, so it is flexible to connect with other backends.

We test the running time of the distributed circuits generated from GD and MT. The testing machine is an AWS m5.large instance with 2vCPU and 8GB memory. For GD, we fix the number of dataset to be 10, the size of each dataset to be 100, and iterate 200 rounds. For MT, we fix the data length to be 256. Table 1 shows with different numbers of machines, the slowest part's running time, and the overall speedup compared to uncut execution time. We also list the effective ratio estimated by the solver. For GD, the effective ratio approximates the real speedup very well; For MT, the effective ratio becomes inaccurate as each chunk's runtime reduces. We think the reason is mainly because the underlying circuits need a constant setup time (around 4 seconds). After subtracting it, the speedup becomes close to the effective ratio.

Acknowledgement

This work is partly supported by NSF awards CCF-2106845, CCF-2131476, CCF-1763399, CNS-2016240, CCF-2019285, CNS-2236819, CCF-2219995, CCF-2318974, CCF-2318975, DARPA under Contract No. HR001120C0087, DARPA and NIWC Pacific under Contract No. N6600121C4018, and the Office of Naval Research (ONR) of the United States Department of Defense through a National Defense Science and Engineering Graduate (NDSEG) Fellowship. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

REFERENCES

[1] [n. d.]. ZKProof Community Reference. https://docs.zkproof.org/reference.pdf.

[2] Shashank Agrawal, Chaya Ganesh, and Payman Mohassel. 2018. Non-Interactive Zero-Knowledge Proofs for Composite Statements. In *CRYPTO 2018, Part III (LNCS, Vol. 10993)*. Springer, Heidelberg, 643–673.

[3] Elli Androulaki, Seung Geol Choi, Steven M. Bellovin, and Tal Malkin. 2008. Reputation Systems for Anonymous Networks. In *PETS 2008 (LNCS, Vol. 5134)*. Springer, Heidelberg, 202–218.

[4] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. 2021. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. In *CRYPTO 2021, Part IV (LNCS, Vol. 12828)*. Springer, Heidelberg, Virtual Event, 92–122.

[5] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles are Practical: A Paradigm for Designing Efficient Protocols. In *ACM CCS 93*. ACM Press, 62–73.

[6] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 459–474.

[7] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge. In *CRYPTO 2013, Part II (LNCS, Vol. 8043)*. Springer, Heidelberg, 90–108.

[8] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct Non-Interactive Zero Knowledge for a von Neumann Architecture. In *USENIX*

Security 2014. USENIX Association, 781–796.

[9] Dan Bogdanov, Joosep Jääger, Peeter Laud, Härmel Nestra, Martin Pettai, Jaak Randmets, Ville Sokk, Kert Tali, and Sandhra-Mirella Valdma. 2022. ZK-SecreC: a Domain-Specific Language for Zero Knowledge Proofs. arXiv:2203.15448 [cs.PL]

[10] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. 2020. Coda: Decentralized Cryptocurrency at Scale. Cryptology ePrint Archive, Report 2020/352. https://eprint.iacr.org/2020/352.

[11] 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 *EUROCRYPT 2016, Part II (LNCS, Vol. 9666)*. Springer, Heidelberg, 327–357.

[12] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. 2020. ZEXE: Enabling Decentralized Private Computation. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 947–964.

[13] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. Recent Advances in Graph Partitioning. *Algorithm Engineering* (2016), 117–158.

[14] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short Proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 315–334.

[15] Matteo Campanelli, Antonio Faonio, Dario Fiore, Anaïs Querol, and Hadrián Rodríguez. 2021. Lunar: A Toolbox for More Efficient Universal and Updatable zkSNARKs and Commit-and-Prove Extensions. In *ASIACRYPT 2021, Part III (LNCS, Vol. 13092)*. Springer, Heidelberg, 3–33.

[16] Matteo Campanelli, Dario Fiore, and Anaïs Querol. 2019. LegoSNARK: Modular Design and Composition of Succinct Zero-Knowledge Proofs. In *ACM CCS 2019*. ACM Press, 2075–2092.

[17] Sunil Chopra and Mendu R Rao. 1993. The partition problem. *Mathematical programming* 59, 1 (1993), 87–115.

[18] Arthur C. Clarke. 2013. *Profiles of the Future*. Hachette UK. Originally 1962.

[19] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. 2022. Improving Line-Point Zero Knowledge: Two Multiplications for the Price of One. Cryptology ePrint Archive, Report 2022/552. https://eprint.iacr.org/2022/552.

[20] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. 2020. Line-Point Zero Knowledge and Its Applications. Cryptology ePrint Archive, Report 2020/1446. https://eprint.iacr.org/2020/1446.

[21] Jonathan Frankle, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel J. Weitzner. 2018. Practical Accountability of Secret Processes. In *USENIX Security 2018*. USENIX Association, 657–674.

[22] Joseph A. Goguen and José Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*. IEEE Computer Society, 11–20. https://doi.org/10.1109/SP.1982.10014

[23] Oded Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press.

[24] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1986. Proofs that Yield Nothing But their Validity and a Methodology of Cryptographic Protocol Design (Extended Abstract). In *27th FOCS*. IEEE Computer Society Press, 174–187.

[25] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[26] Jens Groth. 2010. Short Pairing-Based Non-interactive Zero-Knowledge Arguments. In *ASIACRYPT 2010 (LNCS, Vol. 6477)*. Springer, Heidelberg, 321–340.

[27] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *EUROCRYPT 2016, Part II (LNCS, Vol. 9666)*. Springer, Heidelberg, 305–326.

[28] L Gurobi Optimization. 2020. Gurobi Optimizer Reference Manual (2020). https://www.gurobi.com/

[29] David Heath and Vladimir Kolesnikov. 2020. Stacked Garbling for Disjunctive Zero-Knowledge Proofs. In *EUROCRYPT 2020, Part III (LNCS, Vol. 12107)*. Springer, Heidelberg, 569–598.

[30] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *ACM CCS 2013*. ACM Press, 955–966.

[31] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *CRYPTO (Lecture Notes in Computer Science, Vol. 293)*. Springer, 369–378.

[32] Nicholas Pippenger and Michael J. Fischer. 1979. Relations Among Complexity Measures. *Journal of the ACM (JACM)* 26, 2 (1979), 361–381.

[33] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications* 21, 1 (2003), 5–19.

[34] Yuyang Sang, Ning Luo, Samuel Judson, Ben Chaimberg, Timos Antonopoulos, Xiao Wang, Ruzica Piskac, and Zhong Shao. 2023. Ou: Automating the Parallelization of Zero-Knowledge Protocols. Cryptology ePrint Archive, Paper 2023/657. https://eprint.iacr.org/2023/657 https://eprint.iacr.org/2023/657.

[35] Srinath Setty. 2020. Spartan: Efficient and General-Purpose zkSNARKs Without Trusted Setup. In *CRYPTO 2020, Part III (LNCS, Vol. 12172)*. Springer, Heidelberg, 704–737.

[36] the DARPA SIEVE Program. 2022. SIEVE Intermediate Representation. https://github.com/sieve-zk/ir

[37] Riad S Wahby, Ye Ji, Andrew J Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. 2017. Full accounting for verifiable outsourcing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2071–2086.

[38] 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*. IEEE Computer Society Press, 926–943.

[39] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. 2016. EMP-toolkit: Efficient MultiParty Computation Toolkit. https://github.com/emp-toolkit.

[40] Chenkai Weng, Kang Yang, Jonathan Katz, and Xiao Wang. 2021. Wolverine: Fast, Scalable, and Communication-Efficient Zero-Knowledge Proofs for Boolean and Arithmetic Circuits. In *2021 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 1074–1091.

[41] Howard Wu, Wenting Zheng, Alessandro Chiesa, Raluca Ada Popa, and Ion Stoica. 2018. DIZK: A Distributed Zero Knowledge Proof System. In *USENIX Security 2018*. USENIX Association, 675–692.

[42] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. 2022. zkBridge: Trustless Cross-chain Bridges Made Practical. In *ACM CCS 2022*. ACM Press, 3003–3017.

[43] Yibin Yang, David Heath, Vladimir Kolesnikov, and David Devecsery. 2022. EZEE: Epoch Parallel Zero Knowledge for ANSI C. Cryptology ePrint Archive, Report 2022/811. https://eprint.iacr.org/2022/811.

[44] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. 2020. Transparent Polynomial Delegation and Its Applications to Zero Knowledge Proof. In *2020 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 859–876.

A EXAMPLE

Figure 13 shows a code snippet of the Merkle tree we used in the benchmark. The prover uses a binary Merkle tree to compute the hash of N private data blocks. The `merkle` function computes the hash of a segment of data blocks. Then the verifier calls `verify` to verify the hash. When the tree is a leaf node, `merkle` calls `sha256_leaf` to fetch a data block and hash it; When the tree is a non-leaf node, `merkle` recursively computes its two subtrees' hashes, then use `sha256_node` to compute the merged hash. The input has 4 data blocks, and we cut the code into 2 fragments.

Here we annotate `sha256_node` and `sha256_leaf` as **`atomic`**, and `load_block` as **`plocal1`**. So during the shallow simulation, `sha256_node` and `sha256_leaf` will not be unfolded, and `load_block` will be ignored and executed during the deep simulation.

Shallow simulation executes the program with K_0 knowledge, yielding a sequence as Figure 14. The recursive function `merkle` is unfolded. Each variable is annotated with a unique number to distinguish it from others with the same name. Line 1-5 computes the hash of data block 0 and 1, line 6-11 computes the hash of data block 2 and 3, line 13-14 merges them, then line 15 verifies it.

Notice in Figure 13 `merkle`, the recursion condition is a `pub0` expression. This is necessary for the unfolding to proceed during the shallow simulation phase. If there is any non-`pub0` expression involved in deciding the control flow, then 1) the shallow simulation does not have enough knowledge to evaluate that expression so it can not decide which branch to take, and 2) even if it uses the prover's knowledge to evaluate that expression, the information will be leaked to the verifier via the generated programs.

Then live variable analysis generates a dependency graph as in Figure 15. The ILP/PBO solver decides to cut it between lines 5 and 6, so the only dependency is `hash5`.

Now the compiler can already generate two programs as in Figure 16. We only show the produce and sync functions here. Note `produce_5` in chunk 1 records a digest of all variables that are produced by Figure 14's line 5 for consistency check in the end. Here it is `hash5`. Similarly, `sync_6` records a digest of all variables dependent by Figure 14's line 6 after assigning values to them. But we do not know `hash5`'s value yet, so we mark it using a question

```
#define N 4
/* compute merkle hash of block[left, ..., right-1] */
pvt1 int[8] merkle(int left, int right) {
    if (left + 1 >= right) {
        pvt1 int[8] hash = sha256_leaf(left);
        return hash;
    } else {
        int mid = (left + right) / 2;
        pvt1 int[8] hashL = merkle(left, mid);
        pvt1 int[8] hashR = merkle(mid, right);
        pvt1 int[8] hash = sha256_node(&hashL,&hashR);
        return hash;
    }
}

plocal1 plc1 int[64] load_block(int block_id) { ... }

/* compute the hash of a block */
pvt1 int[8] sha256_block(pvt1 int[64] &data) { ... }

atomic void verify(pvt1 int[8] *hash) { ... }

/* compute the hash of a data block */
atomic pvt1 int[8] sha256_leaf(int block_id) {
    /* prover loads and commits a block */
    plc1 int[64] plc_block = load_block(block_id);
    pvt1 int[64] block = {0};
    for int i = 0; i < 64; i = i + 1;
        block[i] = commit(plc_block[i]);

    pvt1 int[8] hash = sha256_block(block);
    return hash;
}

/* compute the hash of two merged hashes */
atomic pvt1 int[8] sha256_node(
    pvt1 int[8] *hashL, pvt1 int[8] *hashR) {
    pvt1 int[64] block = {0};
    /* merge two hashes */
    for int i = 0; i < 8; i = i + 1; {
        block[i] = hashL->[i];
        block[i+8] = hashR->[i];
    }
    for int i = 16; i < 64; i = i + 1;
        block[i] = 0;

    /* compute the hash of the merged results */
    pvt1 int[8] hash = sha256_block(block);
    return hash;
}

unit main() {
    pvt1 int[8] hash = merkle(0, N);
    verify(&hash);
    return;
}
```

Figure 13: Merkle Tree Example

```
1  pvt1 int[8] hash1 = sha256_leaf(0);
2  pvt1 int[8] hashL1 = hash1;
3  pvt1 int[8] hash3 = sha256_leaf(1);
4  pvt1 int[8] hashR0 = hash3;
5  pvt1 int[8] hash5 = sha256_node(&hashL1, &hashR0);
6  pvt1 int[8] hashL0 = hash5;
7  pvt1 int[8] hash7 = sha256_leaf(2);
8  pvt1 int[8] hashL3 = hash7;
9  pvt1 int[8] hash9 = sha256_leaf(3);
10 pvt1 int[8] hashR3 = hash9;
11 pvt1 int[8] hash11 = sha256_node(&hashL3, &hashR3);
12 pvt1 int[8] hashR2 = hash11;
13 pvt1 int[8] hash13 = sha256_node(&hashL0, &hashR2);
14 pvt1 int[8] hash0 = hash13;
15 verify(&hash0);
```

Figure 14: Merkle Tree after Shallow Simulation

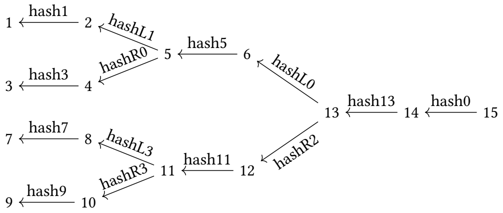


Figure 15: Merkle Tree Dependency Graph

```
1  /* chunk 1 */
2  pvt1 int[8] hash1 = sha256_leaf(0);
3  pvt1 int[8] hashL1 = hash1;
4  pvt1 int[8] hash3 = sha256_leaf(1);
5  pvt1 int[8] hashR0 = hash3;
6  pvt1 int[8] hash5 = sha256_node(&hashL1, &hashR0);
7  produce_5();
8  consistency_check();
9
10 void produce_5() { add_out_digest(2, hash5); }
```

```
1  /* chunk 2 */
2  pvt1 int[8] hash5;
3  sync_6();
4  pvt1 int[8] hashL0 = hash5;
5  pvt1 int[8] hash7 = sha256_leaf(2);
6  pvt1 int[8] hashL3 = hash7;
7  pvt1 int[8] hash9 = sha256_leaf(3);
8  pvt1 int[8] hashR3 = hash9;
9  pvt1 int[8] hash11 = sha256_node(&hashL3, &hashR3);
10 pvt1 int[8] hashR2 = hash11;
11 pvt1 int[8] hash13 = sha256_node(&hashL0, &hashR2);
12 pvt1 int[8] hash0 = hash13;
13 verify(&hash0);
14 consistency_check();
15
16 void sync_6() {
17   hash5 = ?;
18   add_in_digest(1, hash5);
19 }
```

Figure 16: Merkle Tree Generated Programs

mark. The code can already be distributed to individual verifiers, who will count on the provers to supply encrypted hash5 during the runtime. But the provers need to know these values before interacting with the verifiers. So they perform deep simulation on Figure 14's program based on the secret input and find out hash5's value when reaching line 5. Here the type system again guarantees the deep simulation can proceed. For example, suppose there is any K_2 value that is dependent between two chunks, and it relies on the verifier's randomness, then the prover at the compile time can not guess this value, thus it can not proceed.

In the end, chunk 1 and chunk 2 can run in parallel without any communication. When both chunks' execution finish, they will run a consistency check together to make sure the hash5 used in chunk 2 is indeed produced by chunk 1's execution. Here we can see the live variable analysis and the cut searching should find the minimal dependency between the two chunks that guarantees the safe execution of both. Suppose the live variable analysis fails to determine that hash5 is the dependency between chunk 1 and chunk 2, then chunk 2's line 4 will have undefined behavior. Suppose the cut searching algorithm considers hash1, hash3 and hash5 are all needed by chunk2, then we need to commit more data in sync_6, which will slow down the runtime performance.