ACHILLES Resource Estimation Approach and Results

Kinan Dak Albab, Rawane Issa, Peter Flockhart, and Wyatt Howe

{babman, ra1issa, pflock, whowe}@bu.edu

Boston University

September 28, 2020

Abstract

Analysis of resources consumed by programs is an important factor in all aspects of the software development cycle, from early design stages, throughout development and testing, and during maintenance and updates. Additionally, it informs decision-makers on the practicality of software solutions and their cost.

The importance of accurately analysing a program's resource requirements is even greater for secure computation, as it introduces many new dimensions of costs and resources, and diverges from the traditional modes of analysis in plain computation. Accurately reasoning about the performance and resources of secure programs is far less intuitive than for traditional programs. in particular, the added abstractions needed to support secure programming paradigms may hide critical aspects of the concrete costs and resource requirements.

We present Carousels, a generic framework for statically analysing secure programs and estimating their required resources. Our framework combines certain aspects of dynamic analysis and supports user interactions to improve on the accuracy and utility of its output estimates. The framework is generic: it decouples the underlying resource estimation program logic, secure protocols costs, resource metric definitions and the host language type system and syntax from one another. This allows the framework to support new protocols and metrics without the need to modify either the tool logic or implementation.

We discuss several applications and benefits to our tools throughout the development cycle, including design, testing and debugging, and iterative development.

Contents

1 Introduction

Secure Multiparty Computation comes in many different flavors, ranging from secret sharing based approaches to garbled circuits and homomorphic encryption. Each of these approaches is realized by various concrete protocols. These protocols often exhibit trade-offs between various dimensions of costs (e.g. network calls vs CPU usage).

Additionally, operators and primitives provided by these protocols have radically different cost profiles, as operations that are slow in one protocol may be fast in another. The difference is also not necessarily in the constants: the costs of these protocols are dependent on a variety of parameters, some of which are incomparable.

Finally, modern MPC frameworks and languages express functionality using high-level abstractions that aim to hide much of the cryptographic complexity, and attempt to stop developers from "shooting themselves in the foot" by restricting the allowed composition and interaction of these abstractions. While very helpful to beginners, this approach hides many of the underlying costs and interactions that occur during program execution, making it difficult for programmers to reason intuitively about the resources used by their programs as they are developing them.

In fact, the better the abstraction is the more natural it feels within the language, and the more it hides about its resource usage. This is a common theme when designing abstractions through out Computer Science. For example, remote procedure calls (RPC) are a very powerful abstraction that simplifies making network calls, it is implemented in some languages via a completely transparent abstraction that is identical to a local function call (e.g. RPyC in Python). A common pitfall encountered by many distributed systems students is failing to distinguish between local and remote calls, or failing to recognize the costs of using RPC, therefore structuring their code in an in-efficient way (e.g. putting an RPC call inside a loop, instead of making the RPC call once and putting the loop in the remote function).

With MPC, this problem is exacerbated by having generic abstractions: in languages such as Obliv Rust, the MPC program is de-coupled from the underlying protocol used to execute it securely. The program (or parts of it) may be run against different protocols, making the underlying costs even more obscured.

These challenges make machine-assisted resource estimation critical for successfully popularizing secure computation techniques. Programmers should be given good estimates of the resource use of their programs as they are developing them, which they can use to improve the efficiency

of their programs, and make better informed decisions about the design of their programs and underlying protocols. Resource estimates may also be used for testing and debugging, as they can detect certain insecurities and anomalies in the code. Resource estimators may be a role in automatic compilation and optimization of MPC programs, as they can be used to determine which combination of primitive implementations is more suited for the particular program at hand.

Our tool, Carousels, aims to achieve these goals. Carousels is an open source library available publicly at `https://github.com/multiparty/carousels`. Carousels runs as a web application in the browser, it does not need a server component and can be hosted statically, this allows us to build a friendly and easy to use UI, with syntax highlighting, various debugging and user interaction capabilities, as well as visual plots. Carousels also contains a command line API. A demo is available at `https://github.com/multiparty/carousels`.

Resource Usage Metrics in MPC Traditionally, the most important cost of running a program is its running wall time. Historically, this cost was easy to estimate and predicate by considering relatively simple metrics of programs (e.g. big O analysis of program code). As the computer architecture became more sophisticated, the running time of programs became dependent on a variety of metrics, including its memory usage, its friendliness to caching and parallelization, and its susceptibility to the various micro-architectural optimizations that are performed on the program instructions through out the modern pipeline (e.g. out of order execution, branch prediction etc).

Similarly, analyzing MPC programs successfully require consideration of several metrics including its overall network usage (e.g. total number of bits exchanged), the number of network rounds/calls made (many messages may be independent or batched), CPU utilization, memory utilization, and others. Many of these metrics are not merely aggregates over the different instructions in the program, they rather depend on the structure of the program itself. While two sequential programs consisting of n multiplications occuring over some input in some order essentially have very similar performance, they may have radically different costs if run under MPC: consider running such two programs in a secret sharing protocol (e.g. BGW), if these multiplications were structure in sequence (i.e. the output of first is fed into the next), the program will exhibit large network rounds, while if these multiplications were essentially independent (i.e. each multiplication is executed over non-overlapping portions of the input), they will only need to perform a single network call if compiled and executed optimally.

Resource Estimation Requirements These different challenges essentially set our desired requirements for estimating resources used by an MPC program: (1) Estimation must involve as little human intervention as necessary. (2) Estimation must be fast: it cannot be as expensive as running the code. Otherwise, continuously using it for feedback during the development cycle becomes impractical. (3) Estimation must "see through" MPC language abstractions to account for these costs correctly. (4) Estimation must support many protocols. (5) Estimation must support many metrics.

1.1 Static Resource Estimation

We believe static analysis is particularly suitable for resource estimation of MPC programs. In static analysis, the code being analyzed does not need to be run, instead it is parsed and processed via some form of logic or analysis algorithm. This technique is widely used for various program analysis tasks, and we discuss some of the related work in the next section.

Static resource estimation may be performed using a variety of techniques: estimators may rely on program logics that reason about resources, type systems and inference that consider desired resources explicitly within its types, or computer algebra and abstract interpretation techniques that help derive mathematical models of the program and its resource usage.

These various techniques share a common challenge: in full generality, statically determining resource usage of a program is demonstrably undecidable, this can be seen as an instantiating of Rice's theorem. This often leads static analysis researchers to weaken the problem, by allowing some errors in the estimates, or restricting the features of the host programming language (e.g. removing recursion). Thus, trading in decidability for accuracy or expressivity.

The crux of this undecidability (and thus the difficulties and inaccuracies of static analysis) comes from recursion and other form of control flow that depend on user input. It is inherently hard to accurately and completely describe the resource use of some loop, where the loop condition is some expression in the dynamic inputs to the program, without running this loop or having the description be essentially equivalent to (and no more understandable than) the initial program.

However, MPC code is generally oblivious to the values of its inputs. All observable behaviors of the code (including all its resource usage) must only depend on the public constants in it and not on the secret inputs, otherwise it risks leaking information. While there exists protocols that explicitly trades in some form of quantifiable leakage in order to be faster (e.g. differential private ORAM vs ORAM, protocols with one bit leakage, etc), these protocols remain relatively few, and large portions of them remain oblivious.

Concretely, this means that in MPC code, regular non-obliviouss if statements, loop conditions, and recursion base case checks are often dependent on the public constants only. These are the kinds of programs for which static analysis can provide fast and accurate resource estimates.

1.2 Dynamic Resource Estimation

Dynamic resource estimation relies on bench-marking or simulating the program, and measuring its resource consumption. Traditionally, a program is bench marked by running it many times on the same input, and averaging the result, so that the benchmarks normalize over the noise and variance introduced by artifacts in the benchmarking environments.

Generally, MPC programs are slower than their insecure counterparts, and require a more involved setup with various network parties and specific network infrastructure and capabilities. Thus, running the MPC program in a benchmarking environment that is similar to the production environment will result in an expensive and impractical estimation pipeline, and it is impractical to use for continuous feedback during the development process.

Alternatively, an MPC program may be simulated, by running it locally on a single machine within an environment that mimics a networked environment, while monitoring the usage of various resources. This approach is faster than the previous one, but it still requires that the estimation phase be proportional to the actual resource/runtime of the input program, rather than the size of its code.

This approach does not scale well in terms of metrics and protocols: as supporting a new metric may require implementing a custom simulation environment and runtimes. Furthermore some metrics are more difficult to measure this way than others: it is hard to measure the number of network rounds without additional instrumentation and accounting in the input program, since a black box simulator may not distinguish between dependent and independent network calls.

1.3 MPC Software Performance Engineering

Software performance engineering is a well established discipline for development of software systems where performance and resources are critical. This discipline is based on a methodical and scientific approach to optimizations: small changes are made iteratively to existing programs and systems, after which their effects on the performance and resources are measured, and the changes are kept, discarded or further modified according to these measurements.

With the decline of Moore's law, as well as the shift to cloud computing, where the monetary cost of hosting and running programs directly relate to their resource use, and can change drastically with relatively minor optimizations or slow down, software performance engineering is seeing a new reemergence [1].

We believe that software performance engineering is particularly critical to the success of MPC applications, as MPC programs are extremely sensitive in their costs and resources to the any code modifications, and they frequently require optimization and modifications to the conceptual algorithms they are meant to implement, to make it friendlier to the unique cost trade-offs of MPC primitives.

Therefore, we believe that efficient MPC programs can only be achieved via systematic and incremental changes made both on the implementation as well as an algorithmic level, that eventually cultivate into an optimal program. Such methodology may only be carried out given accurate measurements or estimates of a programs resource use.

We believe dynamic measurements attained by running and benchmarking MPC programs are unsuited to play this role. It is both slow and difficult to accurately measure costs of the network operations of MPC programs run via a simulation environment, it is even slower and costlier to do so in an environment similar to the production one, with actual network nodes and network traffic. Additionally, such measurements will suffer the same drawbacks of measurements of modern non-MPC program. Modern deployment environment have a high level of noise and variance, stemming from their reliance on cloud computing, shared hardware and virtualization technology. Such high amount of noise may obscure the full effects of a single small code optimization during measurement [1].

We envision static analysis tools such as Carousels as the ideal approach for MPC performance engineering. Such tools can be integrated into the development cycle of MPC software, and can provide fast feedback to developers and other components of the tool-chain alike, without the cost or complexity of managing a measurements environment, and can be easily adapted to any interesting resource or cost parameter of choice, or underlying protocol and architecture. We demonstrate various benefits from this approach in our applications section, especially in the context of developing solutions for the challenge problems for the GFI document.

2 Related Work

2.1 Resource Estimation Techniques

Static Analysis Techniques Static Analysis has been frequently used to analyze programs in academia and industry. Recently, static analysis for resource estimation has been getting some traction, especially with the rise in popularity of new computing paradigms, such as randomized, cloud and parallel computing, which add new dimensions of resources and costs.

At first, static resource estimation was focused on simple sequential programs with polynomial

costs in the size of the inputs. However, recent advances saw approaches extended to handle probabilistic programs [2], and programs with exponential costs [3]. Particularly relevant to us for the future is work on symbolic and algebraically simplifying recurrences estimating resource usage and finding closed form upper bounds for them [4].

In industry, a number of static analysis tools are developed by major companies, as part of their ecosystem for hosting and providing online services. Resource and cost estimators were developed by IBM [5], Amazon AWS [6], and Google cloud [7].

These tools are becoming very prevalent and popular, and with recent advances in static analysis techniques, as well as implementation frameworks and higher level languages, they are easier to design and build than ever. In an applied programming languages course that two researchers from this team designed and taught last fall at BU, several groups of students built tool for estimating a variety of interesting resources (e.g. AWS S3 Calls, size of pandas data frames) used by simple programs in popular languages (e.g. Python and JavaScript) in the context of cloud computing and data science [8].

Type Systems A complementary approach to statically providing resource estimates, is to develop type systems where resource usage is a first order concern. These systems may rely on user-provided type annotations, or may perform automatic inference for portions of the program.

Recent work have developed type systems capable of automatically reasoning about upper bounds of simple programs with integers and arrays [9], as well as analyzing how two programs resources usage compare to each other relationally [10]. A lot of these developments are applied in the context of functional languages with strong expressive type systems, such as OCaml [11] and ML [12].

Additionally, this line of work can combine well with common program analysis and verification techniques from theorem proving, notably with the Coq theorem assistant, such that the techniques used provide a machine check-able proof demonstrating that its estimates are sound [13], or allow users to interactively carry out or direct such proofs and analysis [14].

Finally, beyond resource estimation, static and interactive techniques have been used to great success for analyzing and verifying security properties of programs and cryptographic protocols, including synthesizing low-level and efficient implementations of cryptographic arithmetic and protocols from high level code [15], and mechanizing the security proof of cryptographic protocols in a variety of security models, including the random oracle model [16], and universal composable security [17]. Our work links up well with such directions of research, as we hope to extend these formal guarantees throughout correctness, security, and resource use, as well as for proving soundness and accuracy bounds for our tool's estimates.

2.2 Static Analysis and Applications

Software Development The usage of static analysis tools as part of software development is becoming more and more popular, as software systems get more complicated, more critical, and more resource intensive.

In particular, such tools are commonly integrated into the software production toolkit at many companies [18][19][20][21], at various stages of the development cycle, from interactive design, debugging, code reviews, and into deployment.

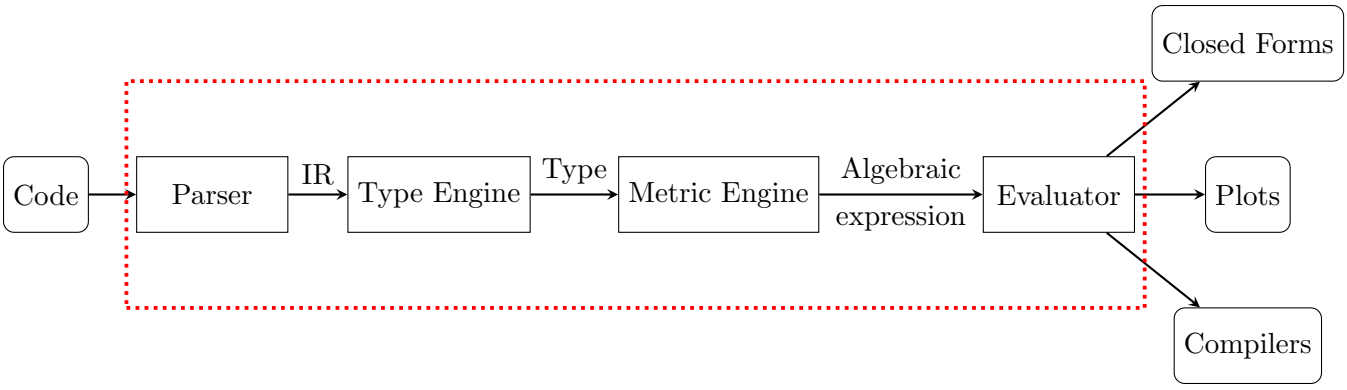

Figure 1: Overview of tool architecture

Cryptography and Security Static analysis tools are commonly used by security experts for detecting and correcting various security risks. Examples include detecting previous vulnerabilities (zero-day attacks) [22], code defects resulting from use of legacy code or library miss-use [23], issues with data sanitization (e.g. SQL injection) and data formatting, as well as detecting potential buffer overflows, timing attacks, and attacks via other possible side-channels [24].

3 Design and Architecture

Overview Our framework, Carousels, is based on static analysis, and thus adapts well to our requirements. Additionally, static analysis allows us to decouple various components in our framework, including the host language type system, metric definition, and protocol cost specification. This means that the core of our framework can be re-used as is between the various metrics and protocols.

Our analysis engine reads the input program, as well as the protocol it is meant to be run with. The protocol is provided as a configuration file specifying the cost model of the primitives it implements. The engine statically analyzes the program, and uses the cost model to transform it into a set of algebraic expressions written in terms of various symbolic variables. These are variables that represent parameters defined in the cost specification (e.g. the security parameter), and sizes of the various inputs.

In the case of a program with no loops, recursion, or regular if statements, these expression will be simple, and can be re-written into a closed form trivially. However, recursive or iterative programs will result in a more complicated set of expressions, which may themselves be recursive or mutually recursive. Furthermore, if the program contains control flow that depends on the value of the input, this control flow is lifted into the algebraic expressions and symbolic variables representing the input values it depends on are added to the expressions. This allows us to use elements of dynamic analysis surgically in portions where we have to.

The resulting algebraic expressions may be evaluated at various plot to produce concrete (i.e. numeric) estimates of resource usage for various inputs sizes and configurations. For simple programs, evaluating these expressions is constant relative to the input program runtime, and only require providing values for input lengths and protocol parameters. For more complicated programs, we rely on memoization and optional user hints to evaluate these expressions efficiently.

3.1 Parsing and Intermediate Representation

Carousels aims to be generic and reusable across protocols, costs, and host languages. Internally, Carousels represents programs using a simple Intermediate Representation (IR) that we defined. The IR contains a simple set of common features between various different languages. These include constructs such as function calls, variable definitions, expressions, conditionals and loops.

Parsers for host languages can be plugged into Carousels, they must read input in the host language and translate it into our IR. We implemented one such parser for Rust. Our IR is mostly typeless, it only specifies types for function signatures (the parameters and return types) when the function is defined. The types for constructs within the body of a function are inferred automatically later in the pipeline.

Being nearly type-less allows the IR to be compatible with a variety of languages, including un-typed dynamic languages, provided that gradual typing frameworks or user annotations are provided for every function definition signature (e.g. TypeScript for JavaScript or Typing hints for Python3).

3.2 Dependent Typing Engine

The cost of a given MPC operation depends on the type of its operands. For example, adding two floating point secret inputs is much more expensive than two naturals. Furthermore, MPC operations whose other operand are constant are usually cheaper than those where both operands are secret.

Carousels infers and tracks the required type information as it analyzes the code, so that it can distinguish between cases similar to the above. Furthermore, types in Carousels are dependent, they may refer to other types or constructs. For example, the type of an array in carousels specifies the type of elements this array has, as well as its length, which may be a known constant, or an expression dependent on the types of other variables or constructs in the program.

Every type in Carousels contains a flag that determines if it is secret or public. Carousels support basic types, including naturals, floating points, arrays, vectors, matrices, strings, and custom abstract (keyword) types. Additionally, Carousels support function types representing user defined functions and closures. Finally, Carousels has two special types: ANY and UNIT. The first is used for constructs for which a type cannot be inferred (e.g. calling some unknown function). The second is used for certain constructs that do not produce values (e.g. a for loop).

Carousels performs type checking and inference for basic language constructs, for example accessing an array must be done with a natural index and returns something of the same type as the element type of that array. These basic constructs are composed and used to assign types to more complicated expressions. However, in modern programs, this is not enough, a typical program has many invocation to standard library functions, as well as external dependencies and libraries.

Carousels accepts custom typing rules and uses them to look up what the type of such an expression should be. These typing rules are specified in a JSON-like configuration format. Each typing rule specifies the kind of IR expression it applies to (e.g. function calls), as well as a pattern to that has to be matched for the rule to apply (e.g. the function name and type of parameters). These patterns can be generic and can quantify over types, and are expressed using familiar Regular expressions, enhanced with additional constructs to simplify common patterns. We show an example type rule in figure 2.

When a typing rule matches a certain IR expression, the full context of that expression, including

```
{
    rule: {
      nodeType: 'FunctionCall',
      // extended regex pattern language
      // this matches any expression on the form <anytype>.clone()
      match: '@T\\.clone\\(\\)'
    },
    value: function (node, pathStr, children) {
      // cloning preserves the type
      return {
        type: children.leftType.copy(),
        parameters: []
      };
    }
}
```

Figure 2: A sample typing rule for the clone standard API in Rust

its node in the parse tree, and the type of its sub expressions, are passed to the custom handler specified by the rule, which then computes the type of the full expression.

Such typing rules are clearly language dependent. Carousels supports passing such rule files externally, so that support for additional libraries and languages can be added without modifying the core engine.

Un-typable Expressions It is important for the functionality of Carousels that every expression get the most accurate type possible, including its dependent component. At the same time, displaying too many errors, especially in peripheral statements that do not affect the resource usage, will make the tool cumbersome to use.

We attempt to strike a balance in our approach: Carousels will throw errors for illegal usage of secret values. Carousels will not throw errors if it could not determine a type of an expression, instead displaying a warning and relying on the ANY type. However, if such untypable expression was used in certain critical locations (e.g. as a parameter to a recursive function or the domain of a loop), Carousels will throw an error. This allows Carousels to produce estimates even if peripheral code could not be typed. In our experience, this is uncommon, and occurs mostly when using external libraries for which custom typing rules were not provided.

Alternatively, an expressions base type may be inferred by carousels (e.g. it is an array), but its dependent component may not be inferred easily (e.g. its length). In this case, the dependent component is represented by a fresh new symbolic parameter that Carousels creates. This symbolic parameter only appears in the algebraic tool output if it affect resource usage of the program, and will require user intervention to resolve.

Static analysis usually faces such inference issues when the dependent type is determined by control flow statements, such as an if condition or a recursive call. For example, an if condition may set an array to be empty in one branch, and non empty in another.

However, the use of type abstractions, described further below, as well as lifting of dynamic expressions, allows us to handle control flow and recursive calls with good accuracy, making fresh

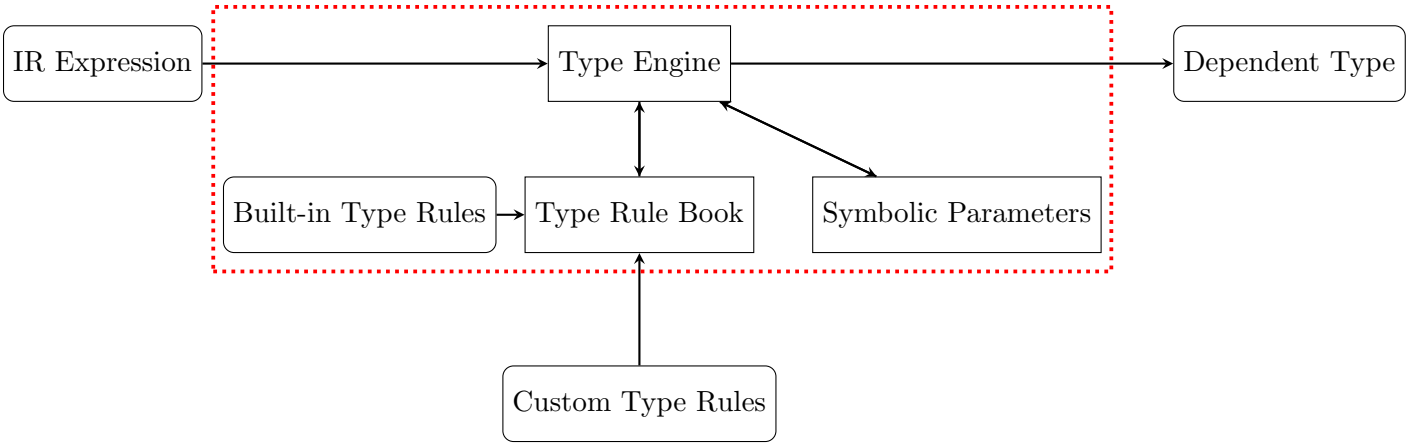

Figure 3: Architecture of typing engine

parameter introduction uncommon.

Lifting of dynamic Expressions Consider the types of the following two variables:

```
let x = [1, 2, 3];
let y = x;
```

The code contains sufficient information to be able to infer the types of x, and y as well as their dependent components. In particular, Carousels will type x as *array(number, 3)* and y as *type(x) = array(number, 3)*.

However, consider the following code:

```
// n has type boolean(v), where v is a symbolic parameter representing its value
let x = [1, 2, 3];
if (n) {
  x = [1];
}
```

The type of x can still be easily inferred to be an array. However, its length is now problematic. One approach to typing x is to take the upper bound, hoping that this will produce an upper bound on the resources used by upcoming portions of the program. However, this is not good enough. The upper bound may be difficult to determine statically, and there is no guarantee this will provide an upper bound, without other modification to the costs logic.

A similar issue occurs in estimating the cost of a conditional branch, should we estimate the cost to be exactly equal to one of these branch, the maximum, or an upper bound on the maximum (say the sum)?

We resolve this by lifting the dynamic condition expression into the dependent type. Dependent types as a theory is powerful enough to handle references to other types, as well as dynamic quantities in them. By using this techniques, and having a sufficiently powerful dependent types language (in our case, arithmetic), we can type x as *array(number, v ? 1 : 3)*.

This surgical use of lifting provides us with the most accurate typing of the expression. However, it introduces dynamic quantities into our statics, which may need to be evaluated or reduced to

```
{
  rule: {
    nodeType: 'DirectExpression',
    // matches multiplying any two numeric secrets together
    match: '<type:number@D,secret:true>\\*<type:number@D,secret:true>'
  },
  value: {
    // Symbolic costs in the number of parties p, and the bit size of the field b.
    'Network Bits': '(p-1)*b',
    'Network Rounds': '1',
    'Logical Gates': '1',
    'Total Memory': '(p+1)*b',
    'Memory Access': 'p+5',
    'CPU': '1'
  }
}
```

Figure 4: Sample cost specification rule for secret multiplication with the BGW protocol

attain numeric values or closed form solutions. In essence, this allows us to surgically combine dynamic analysis with out approach, but only for the portions of the code where it is most needed. Further below, we describe the techniques we use to evaluate and simplify the dynamics in our algebraic expressions, without making our evaluation as expensive as dynamical analysis.

3.3 Cost Specification

Costs are described using cost rules similar to the type rules described above. Cost rules contain the kind of IR expressions it is applicable to, as well as a Regex pattern over its type, it additionally specifies the cost of expressions matching that type, which can be a numeric value, or a symbolic algebraic expression in the parameters from the IR expression context, or global parameters defined by the protocol the rules belong to (e.g. the security parameter). Costs are by default incremental, they are aggregated with the costs of the sub expressions according to the metric logic. However, the cost rule may specify that its cost is absolute, in which case the costs of the sub expressions are ignored.

Each protocol can have many cost rules, which can be composed to describe new rules (e.g. the cost of a mux operation is the cost of a secret addition and a secret multiplication). Additionally, costs may be defined generically in terms of primitive operations without specifying a protocol. For example, primitive matrix operations (matrix multiplications) can be defined in terms of multiplication and addition primitives, without specifying which protocol is used to realize this primitive. Such costs can be applies with any concrete protocol, and carousels will automatically concretize their specified costs accordingly. We show a sample cost specification rule for securely multiplying two secrets with the BGW protocol in figure 4.

Users of the tool can specify which set of cost rules (organized in groups) they want to use to analyze their code, allowing them to compare the resource usage of the same program executed with different protocols.

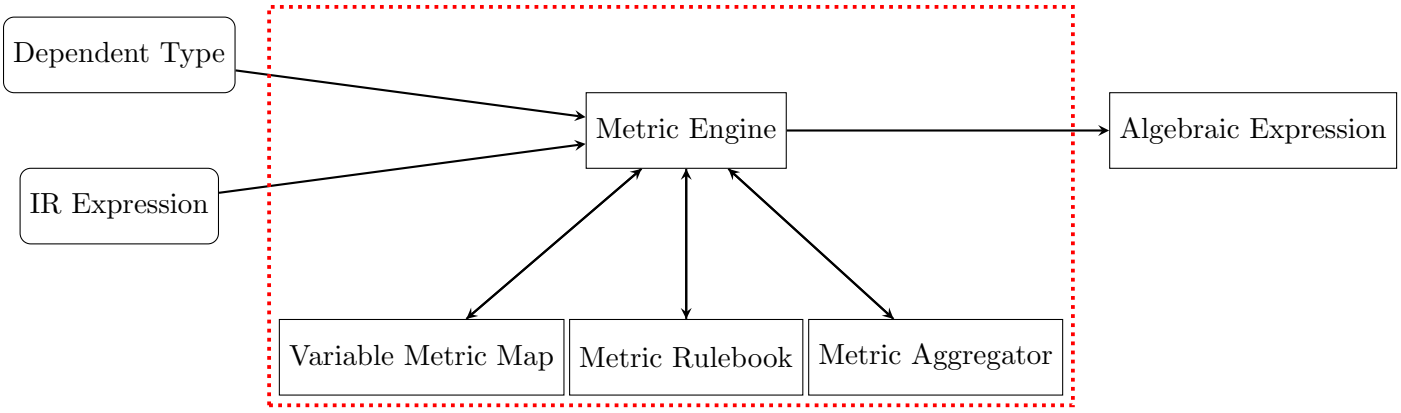

Figure 5: Metric and cost architecture

3.4 Metric Definition

We generally categorize the type of resources we are interested in into two main metrics: a total metric, and a round metric. The first computes the cost of an expression by aggregating the costs of all its sub expressions. Most common resources fall in this category. For instance, the number of memory accesses an addition expression performs is equal to the number of memory accesses require to evaluate its operands plus its own memory accesses.

However, the second category of metrics is rather different. A round metric, such as network calls, or a circuit multiplicative depth (this is important for homomorphic encryption as the size/noise of ciphers is a function of it) does not aggregate globally over the sub expressions. Instead, it is usually the maximum of that metric's value over the sub expression, plus the cost of their composition. For example, the multiplicative depth of expression $(x * y) * (z * t)$ is 2, even though the total number of multiplications is 3.

Carousels provides generic metric aggregation logic for both these categories. These definitions are re-used for every resource of that category, so that no additional code or modifications are required to support new resources. This is especially important because correctly aggregating these costs may be tricky, and re-implementing it for every resource is bug prone.

To simplify implementation, the metric definition is implemented for each category is implemented in a separate modules. The group of cost rules chosen by the user specifies its category, and its corresponding module is loaded by Carousels. These modules provide a single API per IR expression kind, which takes as parameter the types and costs of all the sub expressions, and returns the aggregate.

For example, both modules contain exactly a single API for function calls. The total metric module aggregates by adding the costs of the parameters and context, while the round metric takes the maximum. Similarly, for a code block (such as a function body), the total metric adds up the costs of all the statements, while the round metric takes only the maximum.

This is possible because Carousels keeps track of a scope-aware mapping, that maps every variable to its current cost for a given resource. When a variable appears in an expression, that cost is charged. The metric module specifies what cost should be stored: the total module always stores a zero cost for all variables, since their cost is immediately charged at their creation, this avoids double counting. The round metric stores the cost of the last expression assigned to that

variable, this avoids double counting, since that cost is not charged until that variable is used, resulting in a new cost that is not charged until it is used, and so on until an output expression is reached.

Finally, certain expressions fully consume the cost of a variable. For example, the body of an if statement may only be executed after its condition has been computed. If a variable is used in both the body and condition, and the cost of both is then added, the cost of that variable may have been double counted. Therefore, the cost of the variable used is charged only in the condition. This is not a common case, since plain if statements in MPC program traditionally only refer to public constants (which generally have no round cost), while oblivious if statements work differently and do not require such treatment.

3.5 Overall Engine

The remaining components in the Carousels engine consists of the connecting glue that assembles these previous components together. This glue computes the IR parse tree representation of the program by invoking the parser, and then visits the sub expressions in that representation recursively in post order.

For each visited sub expression, the engine invokes the type engine first, which infers the type of the sub expression via its type inference and typing rules, followed by invoking the metric engine, which aggregates the costs of the sub expression via the metric module, and adds the cost of the expression if any was defined in the cost rules.

This engine is also responsible for managing all state, including scopes and maps for variable types and costs. Whenever a function definition is fully visited, the resulting aggregate cost computed by the engine represents the estimated resource usage for that function. This is assigned to a cost abstraction: a symbolic function that takes costs and dependent type information of the function parameters (such as lengths of parameter arrays) and plugs these values into the expression.

A similar abstraction is created for functions that return types with a dependent component, such as functions returning arrays. Since the length of the returned array may be important to correctly compute the resource usage of other code that calls this function. We refer to this as a type abstraction.

This allows us to handle recursive functions, as well as loops: we unroll loops into as if they are a recursive function, and create a metric and type abstraction to every variable that might have been modified inside the loop. This is similar to deducing a loop invariant. The abstractions at iteration 0 are defined to match the cost and type of these variables prior to the start of the loop, while iteration i abstractions are defined in terms of the previous iteration, and the variables after the loop are defined in terms of the abstraction at the last iteration. Additionally, this allows us to correctly account for round metric costs across function calls.

These cost abstractions represent Carousels' symbolic intermediate output, giving a single symbolic algebraic expression per function, defined in terms of parameters representing its inputs, parameters defined in cost specifications, and any additional symbolic parameters added by the typing engine.

3.6 Concrete Evaluation

The symbolic intermediate output is not sufficient for use by developers or automated compilers. For large programs, the algebraic expressions contain many terms and are too large for humans to parse. Additionally, these expressions may be recursive or co-recursive.

These algebraic expressions can be evaluated: they have a defined set of symbolic parameters, they include only constructs from arithmetic and calls to other expressions. These symbolic parameters consist exclusively of:

1. Global parameters of the protocol (the security parameter, field size, ...).
2. The sizes of the inputs
3. Fresh symbolic parameters introduced by the engine for expressions that were not typeable

The first two types of parameters are important and irreducable: it is expected that the resource use of the code be a function of them. In fact, the relations between them and the resource use is exactly what we want to estimate!

The last type of parameters represents a deficiency in Carousels typing engine, and must be reduced via user intervention. For any given such parameter, the user may provide a value (either numeric or algebraic in the remaining parameters) via the GUI, add additional custom typing rules to address the deficiency generally, or add hint annotations into the input code to instruct Carousels on how to handle the cause of the deficiency in the context of this program only.

After taking care of these fresh parameters (if any). The user need only control the values of the remaining meaningful parameters. The tool evaluates these expressions by plugging in these values directly.

Since every function call and loop iteration translate into a call into an algebraic expression in our intermediate output, it may appear that the run time needed to evaluate these expressions is similar to that of running the program. Commonly, this is not the case. These expressions are only parameterized in terms of the shape and cost of the arguments of their corresponding function calls in the program. Therefore, it is common to have many identical such calls while evaluating the expressions, even though their corresponding calls are unique in the program, making the use of memoization very effective.

For example, consider a function implementing merge sort. It recursively calls itself twice on arrays of half the size. Assuming the size of the array is even, both calls have the same size, and therefore are identical during algebraic evaluation, and therefore can be memoized, even-though they clearly cannot be memoized during the actual runtime. A similar observation can be made about the merge function. This shows that evaluating the expressions of merge sort takes linear time in the size of the input array. We have made similar observations with other examples, where the runtime of the program is exponential but the evaluation is linear time (naive MPC quick sort).

There are conceivably programs where this will not be the case, either because their runtime completely depends on the input values and not just their size, or because they do not have enough function calls to memoize (e.g. a constant time piece of code). In the first case, Our algebraic expression signatures will include additional symbolic parameters representing the input values they depend on (or fresh new parameters if they were untypable), and thus distinct calls during evaluation will not be memoized, which preserves the accuracy of our estimates.

Lifting the dynamics and the use of abstractions trades off accuracy for analysis performance. It allows us to capture programs whose behavior strongly depends on their input values with superb

accuracy, which is traditionally difficult to do statically, but makes our evaluation runtime some non-constant function of the input, which is never more than the code's runtime.

We believe this is a worth while tradeoffs, as such programs are rarely employed in MPC, and memoization has been demonstrated to be effective in curbing the runtime overhead in our experiences for common MPC programs.

For future work, we would also like to produce closed form symbolic solutions. We have several promising ideas on how this can be done. The portions of the algebraic expressions that were lifted from dynamics could be either reduced prior to solving the system via user interaction, or kept in the closed form if appropriate. The crux of this problem is to solve the recurrences (or co-recurrences) in the expressions. This can be done via in lining and rewriting the expressions, and then looking for common patterns that we know how to solve and are frequent (counting loops are one such prevalent pattern), shipping the expressions to Computer Algebra tool or automated theorem prover, or using Master's theorem or similar, if we are happy with asymptotic closed form solutions.

4 Applications

We discuss a few use cases and applications that demonstrate the value of using our tool.

4.1 Secure Program Design

The tool can be used to estimate the resource use for various programs that achieve the same task, or for the same program run with different backend protocols. The estimation may be further specialized according to the exact parameters of the setting instance.

In many cases, MPC programs exhibit interesting trade offs, in part because of having many parameters affect its resource use, and in part because MPC primitives have radically different resource usage, even when their insecure counter parts are very similar.

We provide an example of this, that was studied using an earlier version of our framework and tool: battleships. The game is pretty simple, and in essence it is about matching the positions of certain guesses provided by one party with the positions of the ships chosen by another party. The threat model here is simple, neither players want to trust the others with their data, because they may cheat (i.e. change their ships location after they learn the guesses), so instead they will secret share their data with a collection of servers (e.g. 3 servers) that constitute a battleship game service provider. There are other models that have different security guarantees, and even cryptographic schemes based on commitments that achieve similar results, but they are outside the scope of this use case.

Figure 6 shows the main portion of the battleship code when translated from insecure to MPC naively. The main performance bottleneck in this implementation is the secret equality test in line 23. Equality testing is a lot slower than addition and multiplication in secret-sharing based protocols (e.g. BGW). A popular implementation of equality testing is based on Fermat's last theorem, where the two secrets being compared are subtracted, and the result is raise to the field order (if prime) minus 1 using fast exponentiation.

The performance of this protocol seems to be independent on the size of the square board $n \times n$ at first glance. However, there is an implicit dependence. A guess here is represented as a number/index of the cell in the board. There are n^2 such indices. The field size must therefore

```
// P represents an oblivious protocol trait
// we consider an example with 5 guesses and 5 ships per player
// we assume G and S are arrays containing secret positions of the guesses and ships
let zero = P::run(0);
let a = [zero, zero, zero, zero, zero];
for (i in 0..G.len()) {
    for (s in S) {
        a[i] = a[i] + (G[i] == s);
    }
}
```

Figure 6: Naive translation of insecure battleship into MPC

```
let a = [P::run(1), P::run(1), P::run(1), P::run(1), P::run(1)];
for (i in 0..G.len()) {
    // if guess i matches any of the ships, a[i] will be 0
    for (s in S) {
        a[i] = a[i] * (G[i] - s);
    }
}

// avoids leaking aggregate differences between ships and missed guesses
for (i in a.len()) {
    a[i] = a[i] != 0;
}
```

Figure 7: "Index" version of battleship

be at least n^2, to be able to uniquely represent each position, and the resource use of the secret equality depends on the field size.

The naive implementation can be sped up by trying to reduce the number of secret equality performed, which are currently $|G| \times |S|$. Some of these optimizations may be meaningless if not slower in the insecure world, since the operations they trade in are roughly similar to the existing ones.

We show three potential optimizations, and compare them to each other. The first, **index**, relies on properties of arithmetic operations to perform only $|G|$ tests, the second, **binary**, secret shares a board size array containing 0 for positions where there is no guess (ship respectively) or 1 for positions that do, and thus avoids comparisons all together, and the last, **row-column**, represents each guess as a pair of row and column indices, each index belongs to a domain of size n, rather than n^2 in the naive case, at the cost of needing two equality tests per guess. The code for these three versions is shown in figures 7, **binarybattle**, and **rowbattle** respectively.

It is not easy to compare these three implementations by hand. There are all sorts of hidden correlations between the various parameters (such as the board size and field size). For example, the performance of the row-column version really boils down to the cost of equality testing, for

```
// N is the size of the board
let a = vec::with_capacity(N*N);
for (i in 0..N*N) {
  // logical AND
  a[i] = G[i] * S[i];
}
```

Figure 8: "Binary" version of battleship

```
let a = [P::run(0), P::run(0), P::run(0), P::run(0), P::run(0)];
for (i in 0..G.len()) {
    let (g_row, g_col) = G[i];
    for (s in S) {
        let (s_row, s_col) = S[s];
        a[i] = a[i] + (g_row == s_row) * (g_col == s_col);
    }
}
```

Figure 9: "Row-column" version of battleship

example, if the primitive implementation for equality testing is quadratic in the field size for the resource at hand, then the row-column version wins $2*(\frac{n}{2})^2 = \frac{n^2}{2} < n^2$, additionally, if the reduction in field size is enough to make the other operations perform sufficient better, so that overall there is still improvements even if the cost from doing two equality tests is increased, then the version will still see improvements.

We plot the costs of the three implementations for two important resources: total network messages (each message is of fixed size here, so this a proxy to the number of bits sent in total), and number of network rounds. In our plots, we set the board size to be 8, and the field size to be the next smallest prime 11. We show these plots in figure 10.

Note how the protocols differ in efficiency between rounds and number of messages. This is important. A program with low round complexity and high number of messages contain very few logical dependencies between operations, and is there for more susceptible to parallelization, while the other may be better when run sequentially. Additionally, plotting against other parameters (e.g. board/field size) will show more trade-offs.

Finally, we would like to note that a large portion of the battleship use case, including writing, testing, deploying, and analyzing the different battleships programs, was carried out by a visiting high school intern in our lab. This demonstrates how proper design of MPC languages, as well as proper support from automated tools and IDEs can lower the barrier of entry for non-experts into MPC significantly.

4.2 Testing and Debugging

Carousels may be used to test and debug various MPC implementation for security, performance and functionality defects. An important aspect of secure programs is its constant resource usage, in terms of the input secrets. The program should not leak information about the secrets via

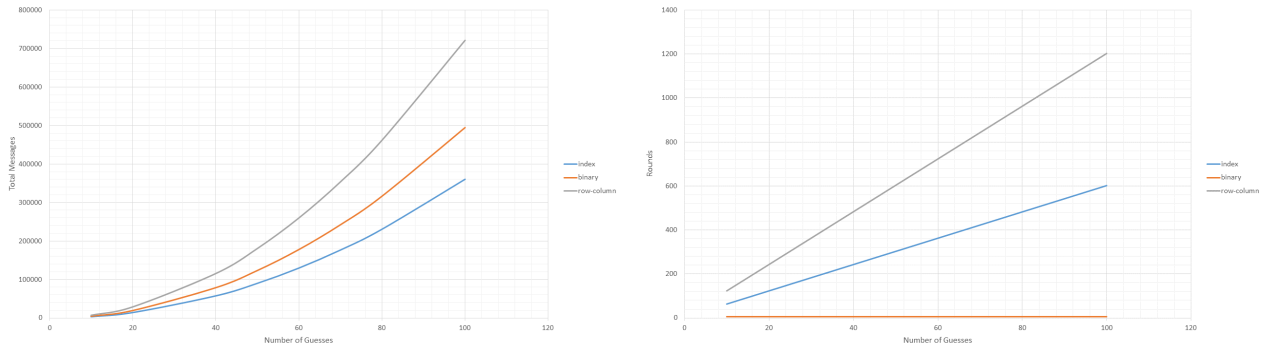

Figure 10: Total number of messages (left) and network rounds (right) of each battleship program against the number of guesses

its memory, time, or network usage. This includes things such as explicitly revealed intermediate values, array sizes, and recursive call counts.

While MPC languages such as obliv rust guarantee that no such leakage occurs unless via an explicit reveal. Programmers may still shoot themselves in the foot by abusing these reveal statements. In many cases, a reveal may be desired because it improves performance while not leaking any information (for example, revealing a masked value), even though they provide opportunities for misuse.

Additionally, MPC programmers may be willing to perform some trade offs between privacy and performance, leaking some information for the sake of speed. However, it is hard to quantify exactly how much information is leaked, and whether the program actually leaks more than it is thought to via indirect means.

We have encountered an issue along these lines while writing our solutions to the GCD challenge problems. Consider the perfectly secure protocol in figure 11. This is a recursive implementation of gcd that is a direct naive translation of the non-MPC algorithm. The base case of the recursion is reached when a becomes 0.

Since a is a secret, the condition $a == 0$ becomes a secret itself, so it is natural to first attempt to use an obliv if to test this condition. However, this makes the program non-terminating, at every invocation of the obliv if statement, including the base case, both branches have to be executed, including a recursive call, meaning that the program never terminates.

Our tool correctly identified this problem, and showed the resource usage of this program to be infinite. Specifically, the symbolic intermediate output created by our tool to represent resource use of the gcd function has no base case, and when evaluating it, quickly causes a stack overflow exception that is handled and displayed to the user by our tool.

Naturally, one attempt to fix this non-termination is to use a regular if statement. Only a single branch is executed at any point in this recursion, causing the program to certainly terminate. However, this requires that the condition be explicitly revealed, as shown in figure 12. This may seem harmless at first, we are only leaking whether a is equal to zero or not. However, a closer look reveals a more dangerous situation, this value is revealed at every recursive call, and the resource use of the function depends on it, since it determines when the recursion terminates.

We analyzed this code using Carousels which plotted the various resource estimates for the

```
fn gcd<P: Obliv>(a: Possession<BigInt, P>,
                 b: Possession<BigInt, P>)
    -> Possession<BigInt, P>
{
    let condition = (b == P::run(0));
    obliv if  {
        a
    } else {
        gcd(b, &a % &b)
    }
}
```

Figure 11: Non-terminating GCD program

```
fn gcd<P: Obliv>(a: Possession<BigInt, P>,
                 b: Possession<BigInt, P>)
    -> Possession<BigInt, P>
{
    let condition = (b == P::run(0));
    if condition.reveal() {
        a
    } else {
        gcd(b, &a % &b)
    }
}
```

Figure 12: Insecure GCD program

different resources against the values of the secret inputs (which is only possible because of lifting). We show this plot for the total number of bits sent over the network, with $a = 20$ and $b \in [0, 100]$, two parties and field size in 2^32 in figure 13.

Secure code should have a constant plot that shows no dependence on the secrets. However, the plot for this insecure code was quite different. It showed that the resource usage changed periodically over values of a or b when the other value is fixed. Careful analysis of the plot demonstrates that the code essentially leaks $a \ mod \ b$. We verified this via manual algorithm analysis of the insecure code.

4.3 Iterative Development

Carousels and other static reasoning tools should be used by developers iteratively during the development process. This saves on valuable development time, and allows developers to get incremental feedback as they proceed through their development.

We have had first hand experience with us as we were working on the challenge problems stated in the GFI document. In particular, merge sort de-duplication and quick sort. Merge and de-duplicate function was implemented initially as a recursive function, the function had three

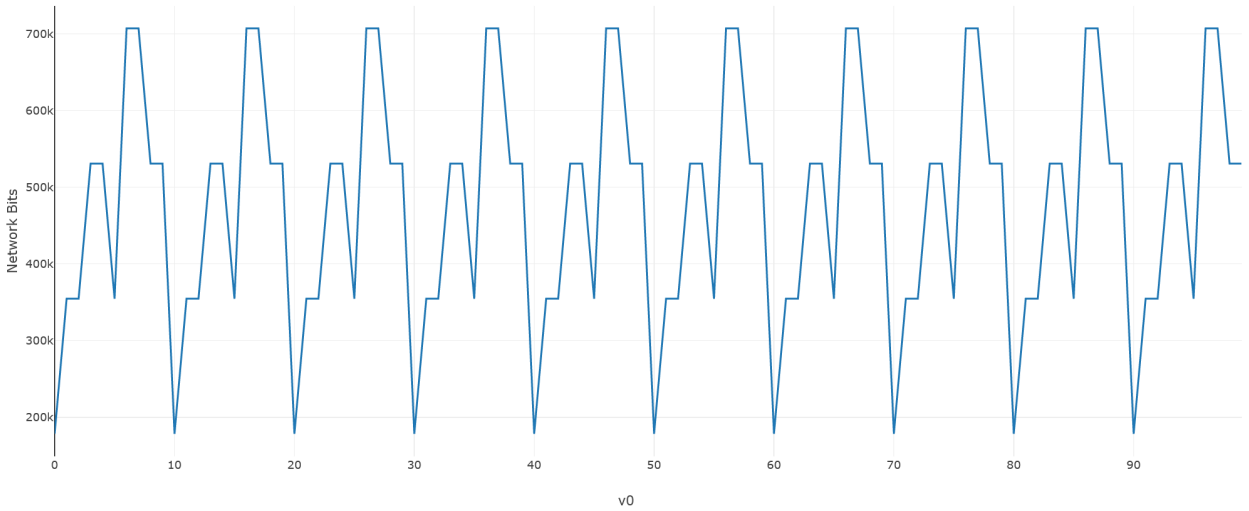

Figure 13: Network bits for insecure GCD program against secret values

recursive cases depending on which of the two current elements is larger, or if they are equal. This yields a linear running time algorithm in the clear. However, because these conditions are secret, they should be used with oblivious if statements, to avoid leaking their values, which by definition means that every branch must be run regardless of the condition that was true.

Analyzing this code through our tool quickly demonstrated that it required exponential resources to run, since every recursive call on arrays of size n, required three recursive calls on arrays of size $n - 1$. The tool was still able to analyze and evaluate the resource use for this program on reasonably large array sizes, due to memoization. The Achilles team at Northeastern revisited the problem, and made incremental changes to the program, until our team was able to verify that the resource use is now quadratic using our tool.

A similar story occurred with quick sort. The team started with an insecure solution, that leads to revealing information about the input arrays indirectly via the sizes of the various partitions and recursive calls made after the pivot is chosen. The team incrementally transformed the insecure program into a secure (albeit exponential) one. We have had similar experiences with other programs who had corner cases where they do not terminate, and the tool helped us detect and address these cases.

References

[1] C. E. Leiserson, "The resurgence of software performance engineering," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, pp. 53–53, 2018.

[2] V. C. Ngo, Q. Carbonneaux, and J. Hoffmann, "Bounded expectations: Resource analysis for probabilistic programs," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), p. 496–512, Association for Computing Machinery, 2018.

[3] D. M. Kahn and J. Hoffmann, "Exponential automatic amortized resource analysis," in *Foundations of Software Science and Computation Structures* (J. Goubault-Larrecq and B. König, eds.), (Cham), pp. 359–380, Springer International Publishing, 2020.

[4] E. Albert, P. Arenas, S. Genaim, and G. Puebla, "Automatic inference of upper bounds for recurrence relations in cost analysis," in *SAS*, 2008.

[5] IBM, "Resource estimation." Accessed: 04-30-2020.

[6] Amazon, "How do i estimate how much my planned aws resource configurations will cost?." Accessed: 04-30-2020.

[7] Google, "Google cloud pricing calculator." Accessed: 04-30-2020.

[8] "Cs591l1: Embedded languages & frameworks." `https://kinanbab.github.io/CS591L1/website/`, Fall 2019.

[9] J. Hoffmann and Z. Shao, "Type-based amortized resource analysis with integers and arrays," in *Functional and Logic Programming* (M. Codish and E. Sumii, eds.), (Cham), pp. 152–168, Springer International Publishing, 2014.

[10] E. Çiçek, G. Barthe, M. Gaboardi, D. Garg, and J. Hoffmann, "Relational cost analysis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), p. 316–329, Association for Computing Machinery, 2017.

[11] J. Hoffmann, A. Das, and S.-C. Weng, "Towards automatic resource bound analysis for ocaml," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), p. 359–373, Association for Computing Machinery, 2017.

[12] J. Hoffmann, K. Aehlig, and M. Hofmann, "Resource aware ml," in *CAV*, 2012.

[13] Q. Carbonneaux, J. Hoffmann, T. Reps, and Z. Shao, "Automated resource analysis with coq proof objects," in *Computer Aided Verification* (R. Majumdar and V. Kunčak, eds.), (Cham), pp. 64–85, Springer International Publishing, 2017.

[14] J. McCarthy, B. Fetscher, M. New, D. Feltey, and R. B. Findler, "A coq library for internal verification of running-times," in *Functional and Logic Programming* (O. Kiselyov and A. King, eds.), (Cham), pp. 144–162, Springer International Publishing, 2016.

[15] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala, "Simple high-level code for cryptographic arithmetic-with proofs, without compromises," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 1202–1219, IEEE, 2019.

[16] A. Stoughton and M. Varia, "Mechanizing the proof of adaptive, information-theoretic security of cryptographic protocols in the random oracle model," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pp. 83–99, IEEE, 2017.

[17] R. Canetti, A. Stoughton, and M. Varia, "Easyuc: Using easycrypt to mechanize proofs of universally composable security," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pp. 167–16716, IEEE, 2019.

[18] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Commun. ACM*, vol. 62, p. 62–70, July 2019.

[19] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, p. 58–66, Mar. 2018.

[20] Microsoft, "Overview of source code analyzers." Accessed: 04-30-2020.

[21] IBM, "Minimizing code defects to improve software quality and lower development costs." Accessed: 04-30-2020.

[22] IBM, "Static and dynamic testing in the software development life cycle." Accessed: 04-30-2020.

[23] Microsoft, "Walkthrough: Use static code analysis to find code defects." Accessed: 04-30-2020.

[24] J. Hoffmann, "Automatic resource bound analysis and linear optimization." Accessed: 04-30-2020.