Ramparts: A Programmer-Friendly System for Building Homomorphic Encryption Applications

DAVID W. ARCHER, Galois, Inc.
JOSÉ MANUEL CALDERÓN TRILLA, Galois, Inc.
JASON DAGIT, Galois, Inc.
ALEX J. MALOZEMOFF, Galois, Inc.
YURIY POLYAKOV, New Jersey Institute of Technology
KURT ROHLOFF, New Jersey Institute of Technology
GERARD RYAN, New Jersey Institute of Technology

Homomorphic Encryption (HE) is an emerging technnology that enables computing on data while the data is encrypted. A major challenge with homomorphic encryption is that it takes extensive expert knowledge to design meaningful and useful programs that are constructed from atomic HE operations.

We present Ramparts to address this challenge. Ramparts provides an environment for developing HE applications in Julia, a high-level language, the same way as "cleartext" applications are typically written in Julia. Ramparts makes the following three contributions. First, we use symbolic execution to automate the construction of an optimized computation circuit where both the circuit size and multiplicative depth are chosen by the compiler. Second, Ramparts automatically selects the HE parameters for the generated circuit, which is typically done manually by an HE expert. Third, Ramparts automatically selects the plaintext encoding for input values, and performs input and output data transformations. These three operations are not easily performed by programmers who are not HE experts. Thus, Ramparts makes HE more widely available and usable by the the population of programmers.

We compare our approach with Cingulata, the only previously known system that automatically generates circuits for HE computations. The HE circuits generated by Ramparts are significantly more efficient than the circuits compiled by Cingulata. For instance, our runtimes for key generation/circuit compilation and all online operations are more than one order of magnitude lower for a sample image processing application used for performance evaluation in our study.

Additional Key Words and Phrases: Homomorphic encryption; Automatic arithmetic circuit generation; Lattice-based cryptography; Usable security and privacy

1 INTRODUCTION

Homomorphic Encryption (HE) is an emerging technnology that enables computing on data while the data is encrypted. For example, HE provides cryptographic means for a medical provider to encrypt sensitive regulated data such as an X-Ray image, send the encrypted image to a third party for the application of proprietary image processing techniques, and return an encrypted result to the medical provider that the provider can decrypt and use to improve the treatment of patients.

Over the past several years, HE has become increasingly performant and practical for real-world applications [10, 19, 22, 32]. Multiple HE libraries such as HELib [21], Palisade [30], and SEAL [25] are now available, and application-focused competitions such as iDASH foster development of efficient HE solutions to real-world problems [23]. However, designing, writing and deploying efficient software that securely incorporates HE remains a major practical challenge: it takes extensive expert knowledge to design meaningful and useful programs that are constructed from atomic HE operations. This knowledge is unfortunately unavailable to most programmers. In

Authors' addresses: David W. Archer, Galois, Inc. dwa@galois.com; José Manuel Calderón Trilla, Galois, Inc. jmct@jmct.cc; Jason Dagit, Galois, Inc. dagit@galois.com; Alex J. Malozemoff, Galois, Inc. amaloz@galois.com; Yuriy Polyakov, New Jersey Institute of Technology, polyakov@njit.edu; Kurt Rohloff, New Jersey Institute of Technology, rohloff@njit.edu; Gerard Ryan, New Jersey Institute of Technology, gwryan@njit.edu.

particular, poorly written HE programs are likely to suffer from computational and storage inefficiency. Another challenge is the difficulty of integrating HE into existing applications without substantial re-factoring and deep technical knowledge of HE fundamentals. Expecting application programmers to re-write existing functions such as image processing or statistical analytics is unrealistic. Expecting production programmers to learn the complex cryptography skills required to implement HE is also unrealistic.

The first goal (G1) of our work is to explore the degree to which we can automate retrofitting of key portions of existing programs with HE. We find that such automated retrofitting can be enabled by abstractions that instantiate HE with few or no hints from programmers or changes to existing code. A second goal (G2) is to leverage those abstractions to automatically transform program constants and variables of diverse data types for interoperability between "in the clear" code and HE.

Software abstraction typically relies on automatic reification into implementations on underlying computational substrates. Programming languages, associated compiler toolchains, and suitable libraries of primitive operations typically provide such reification. Therein lies another substantial aspect of the HE programming challenge we aim to address. Just as it is unrealistic to expect production programmers to extensively re-factor existing applications or learn cryptography, it is also unrealistic to expect them to adopt new toolchains simply to access HE capabilities. Programmers come to trust languages, compilers, and libraries over periods measured in years. Typically, adoption occurs only if new tools provide substantive productivity gains in meeting application functional and performance requirements. Accordingly, our third goal (G3) is to achieve goals G1 and G2 without modifying current compilers.

One challenge in compiling imperative programs for HE is that HE operates by evaluating arithmetic (or, in some cases, Boolean) circuit representations, while general-purpose computation relies heavily on dynamic memory structures and persistent state. This conflict creates a challenge for a compiler targeting HE. One way to resolve this challenge is to convert the imperative program into a functional specification of the computation to be performed. Thus, our final goal (G4) was to explore how one such principled approach, symbolic execution, could be used to generate optimized arithmetic circuits for a rich set of expressions amenable to HE.

The system presented in our paper is a step towards building an ecosystem that enables the development of a broad spectrum of secure distributed applications employing advanced cryptographic techniques to:

- Enable application developers to incorporate new and existing cryptographic computing concepts into their software, while minimizing the cryptographic expertise required of the application developers.
- Enable the system architects and/or developers to be able to express the security and privacy properties and operational environment as desired properties of the system, and automatically receive feedback on the feasibility and costs of these properties.
- Enable advances in the security and performance of cryptographic operations.

Taken together, Ramparts provides a framework for developers to explore the space of distributed applications and possible compositions of different cryptographic techniques. This makes HE technologies more deployable in practical protocols, with easier deployment on distributed systems in a more automated manner.

1.1 Our Contributions

The main contribution of our work is Ramparts, a programmer-friendly system for developing HE applications in Julia the same way as “cleartext” applications are typically written. The system consists of the front-end and back-end components.

The front-end consists of an interactive scientific computation system, Julia, and a symbolic simulator, Crucible. The workflow automatically generating optimized arithmetic circuits includes the following steps:

- The front-end converts Julia functions into symbolic expressions using Julia compiler’s meta-programming capabilities.

```
function sharpen(image::Array{Int,2})::Array{Int,2}
  weight = [[1  1 1]; [1 -8 1]; [1  1 1]]
  image2 = deepcopy(image)
  dx,dy = size(image)
  for x = 2:dx-1, y = 2:dy-1
    value = 0
    for j = -1:1, i = -1:1
      value += weight[i+2,j+2] * image[x+i,y+j]
    end
    image2[x,y] = image[x,y] - (value >> 1)
  end
  image2
end
```

Fig. 1. Implementation of 8-neighbor Laplacian sharpening algorithm in Julia.

- Crucible compiles the symbolic expressions into arithmetic circuits by unrolling loops, removing branches, and removing side-effects.
- The resulting arithmetic circuit is then analyzed: the multiplicative depth is computed and the plaintext encoding is determined. Next, high-level transformations, such as specialized gate selection and circuit-level optimizations, are applied before generating the final circuit.

The front-end also features inline HE using Julia macros and a mechanism for testing and verifying the programs.

The back-end includes Palisade, a lattice cryptography library, that loads the circuits and evaluates them. As part of this work, we added a new "circuit" module to Palisade that supports evaluating the arithmetic circuits generated by Crucible. The circuit module parses the circuit into its corresponding graph representation, which allows for additional (backend-specific) performance analysis, optimizations, and graph restructuring.

Another distinctive feature of our system is data interoperability between the front-end and back-end. Ramparts automatically transforms input data structures into simple operand vectors, and output vectors of such computations (after decryption) into data structures expected by the remainder of the application program. In current systems, such transformations are performed manually.

2 SYSTEM ARCHITECTURE

In this section we first introduce a setting where HE must be used by programmers who are not expert cryptographers. This setting includes both secure (HE) computation and non-secure computation, as well as the need to interoperate between the two. Next, we describe the architecture of our Ramparts system and how that architecture supports this workflow.

2.1 A Programmer's Perspective

In our setting, an analyst (call her Alice) is tasked to *sharpen* an encrypted image and then further operate on the sharpened (unencrypted) result. Such settings arise for example in the analysis of satellite imagery, where a satellite may encrypt an image, an untrusted ground station may perform initial processing on that image, and then an analyst may further process the (plaintext) result on a secure workstation. HE technology comes into play in the second of these three steps: outsourcing processing of sensitive data to an untrusted resource.

Alice writes a program that includes a typical 8-neighbor Laplacian sharpener function called `sharpen`, and wants to run it remotely at an untrusted server on an encrypted 8-bit grayscale image, encoded as an array of pixel values. As shown in Figure 1, `sharpen` iterates over each pixel in the source image (except the outer

```
1  #server code for image sharpening computation
2  using Fhe
3  Fhe.read("fhe.pk", Fhe.KEY_TYPE_PUBLIC)
4  ct = Fhe.read("encrypted-image.ct")
5  res = Fhe.evaluate(sharpen, (ct,))
6  Fhe.write("res.ct", res)
7
8  #client code for decrypting the result
9  using Fhe
10 Fhe.read("fhe.sk", Fhe.KEY_TYPE_SECRET)
11 res = Fhe.read("res.ct")
12 img = Fhe.decrypt(res)
13 save("image.png", img)
```

Fig. 2. Julia code for running `sharpen` on an encrypted image by server and decrypting the result by client.

bounding pixels), modifying each pixel value to be a simple function of its original value and the values of its eight neighbors. Alice remotely runs `sharpen` on the untrusted server using Ramparts as shown in Figure 2.

Alice's program first imports our `Fhe.jl` Julia package (Line 2), which exposes an API for cryptographic key generation, encryption, decryption, and homomorphic function evaluation. Alice then loads public evaluation keys (Line 3) and ciphertext `ct` using the `Fhe.read` function (Line 4), and then calls `Fhe.evaluate(sharpen,(ct,))`, which evaluates `sharpen` homomorphically on the encrypted image[1] (Line 5). The result is then serialized on the server and sent to the client that Alice is using.

On her client machine, Alice loads the secret key (Line 10) and ciphertexts (Line 11), and decrypts the image using `Fhe.decrypt` (Line 12). The decrypted result image is then available (without further programming effort) for use in Alice's continuing program (Line 13).

An important aspect of Ramparts is that the Julia function provided as input (e.g., `sharpen` in the above example) is a *standard* Julia function, programmed as one would write it normally in Julia and containing *no* HE-specific annotations or syntax. We do not ask the programmer to write in a domain-specific language or other restricted programming environment. This approach makes HE easily available to analyst-programmers, and also allows such users to re-use functions on both sensitive and non-sensitive data. However, we do note that there are some limits to the support of Julia in the current version of Ramparts; for example, Ramparts focuses on arithmetic computations supported by our underlying HE library, Palisade. Thus operators such as comparisons are not currently supported in Ramparts.

2.2 Ramparts Architecture

Figure 3 illustrates the architecture of Ramparts. The Julia environment shown at left is used to develop applications, including those functions or in-line expressions to be evaluated on sensitive inputs using HE. Julia is also the platform where non-secure operations are executed. Ramparts uses the Julia compiler's meta-programming capabilities to extract Julia's underlying intermediate representation of the relevant HE code and convert it into an *s-expression* formulation, expanding as needed those operators not natively supported by our compiler. Ramparts then passes this s-expression to the Crucible symbolic simulator. Crucible symbolically executes the function, unrolling all loops and branches, producing a symbolic expression as output. This expression is then automatically converted into an arithmetic circuit. That circuit is inserted back into the compilation

[1]The reader may notice that because key generation was done *before* the function was specified. For this to work in the general case requires the use of *fully* homomorphic encryption. In reality, key generation assumes a specific multiplicative depth of computation, which is either specified explicitly by the user or computed based on a particular function. We discuss this further in §4.

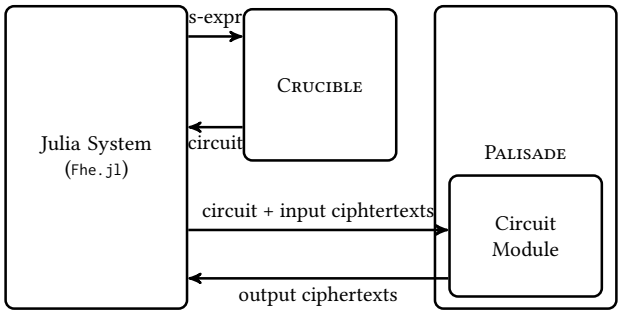

Fig. 3. The Ramparts architecture. `Fhe.jl` first converts Julia code into its s-expression form and sends this to the Crucible symbolic simulator, which returns an arithmetic circuit representation of the code. It then sends this, along with the encrypted inputs, to the Palisade HE library, which evaluates the circuit on the inputs, returning encrypted outputs.

process, where it is scheduled for evaluation. At evaluation time, the circuit (along with suitably encrypted input data) is passed to Palisade, our backend HE library shown at right, for evaluation. We implemented support within Palisade for taking this circuit description as input and making the appropriate HE evaluation calls. The (encrypted) output is then passed back to the Julia runtime where it can be decrypted (again using Palisade), reconstructed into the expected result data type, and integrated into the remainder of the running Julia program.

One difficulty in using HE is that computational effort is highly correlated to both overall size and multiplicative circuit depth of the function to be evaluated, and programmer effort in current systems is highly correlated to optimizing circuits to reduce computational effort. Prior work typically relies on manual construction of circuits and manual minimization of circuit size and depth. In Ramparts, our use of symbolic execution automates the process of circuit construction and optimization. We automatically implement optimizations such as sub-expression elimination, constant folding, and partial evaluation optimizations as a side effect of symbolic execution, thus assuring that the circuit has minimum size. In addition, Ramparts optimizes to reduce multiplicative circuit depth.

Another difficulty in using existing HE is that manual HE parameter tuning is a complex process that requires significant cryptographic expertise. In Ramparts, we automatically extract circuit depth and other information, and use it to automatically select the HE parameters.

A third substantial difficulty in integrating HE into applications is that input data structures to HE computations must first be transformed into simple operand vectors, and output vectors of such computations must then be transformed (after decryption) into data structures expected by the remainder of the application program. In current systems, such transformations are performed manually – an effort-intensive process. In Ramparts, we automatically perform such input and output conversions. In addition, Ramparts automatically chooses appropriate encodings for input values. Thus Ramparts integrates HE functions seamlessly into Julia programs not only from an execution scheduling perspective, but also from a data interoperability perspective.

While Ramparts provides several such usability improvements over existing HE systems, it does so with little performance penalty. In §6 we provide a detailed performance evaluation, comparing Ramparts to three alternative HE systems in terms of capabilities and overall system performance. We find that Ramparts is quite competitive with even handwritten implementations that directly call the underlying HE library API.

3 PRELIMINARIES

In this section, we describe homomorphic encryption (§3.1) and briefly discuss the Julia programming language (§3.2).

3.1 Homomorphic Encryption

A *(public-key) homomorphic encryption (HE) scheme* is a public-key encryption scheme with an additional Evaluate operation, which takes as input a public key, a function f from some circuit class $\mathcal{F}$, and a set of ciphertexts $\{c_i\}$, and outputs the encryption(s) $\{c'_j\}$ of f applied to those ciphertexts.

More formally, we define a homomorphic encryption scheme $\mathcal{E}$ over ring R as a tuple (KeyGen, Encrypt, Decrypt, Evaluate) of four algorithms defined as follows:

- KeyGen($\lambda, \mathcal{F}$): Takes as input security parameter λ and circuit class $\mathcal{F}$, and outputs a keypair (pk, sk).
- Encrypt(pk, m): Takes as input public key pk and message $m \in R$, and outputs ciphertext c.
- Decrypt(sk, c): Takes as input secret key sk and ciphertext c, and outputs message m.
- Evaluate(pk, $f, (c_1, \ldots, c_n)$): Takes as input public key pk, circuit $f : R^n \rightarrow R^m$ where $f \in \mathcal{F}$, and ciphertexts $c_1, \ldots, c_n$, and outputs ciphertexts $c'_1, \ldots, c'_m$.

An HE scheme is *correct* if decrypting the output of Evaluate equals the output of the circuit applied directly to the corresponding plaintext inputs. We say that $\mathcal{E}$ is (1) *homomorphic for circuit class* $\mathcal{F}$ if correctness holds for all $f \in \mathcal{F}$, (2) *fully homomorphic* if the circuit class $\mathcal{F}$ comprises all (polynomial-sized) circuits, and (3) *depth-d leveled homomorphic* if the circuit class $\mathcal{F}$ comprises all circuits of (multiplicative) depth d. In this work we focus on leveled homomorphic encryption.

By "circuit class", we mean *arithmetic circuits* over some fixed ring R. A circuit is defined as a directed acyclic graph with each node being either an *input node* (a node with no input edges and labeled with some symbol 'x_i') or an *evaluation node*, which takes one or more input edges and has one output edge. We currently support five operations: $+$, $-$, $\times$, $\gg$, and $\cdot$, for addition, subtraction, multiplication, right shift, and dot product, respectively.

In addition to these five operators, HE schemes also introduce scheme-specific "maintenance" operations to control the noise growth and reduce the ciphertext size or dimension. Common examples include Relinearize, ModReduce, and Recrypt. The Relinearize operation reduces the dimension of ciphertexts after homomorphic multiplication, the ModReduce operation reduces the noise growth in leveled HE schemes, and the Recrypt operation "refreshes" the noise in fully homomorphic schemes so that more computations could be performed.

While in theory any function can be compiled into an arithmetic circuit, in practice this conversion may not be efficient. There are two particular issues at hand:

(1) The first difficulty relates to looping over a variable. If the variable's value is not known at compile time, the compiler must set some upper bound on the loop and unroll the loop up to that bound, potentially resulting in unnecessary executions of the loop at runtime if the actual bound is smaller than the assumed bound. Moreover, looping N times may "blow up" both the compilation time and circuit size by as much as $N\times$.

(2) The second difficulty relates to the particular ring we choose for computation. By using a boolean ring, we can support comparisons on encrypted data. However, most practical HE schemes encode the plaintext data using the cyclotomic ring $R_t = \mathbb{Z}_t[x]/\Phi_m(x)$, where t is a plaintext modulus that should be large enough to guarantee the correctness of plaintext computations and $\Phi_m(x)$ is an m-th cyclotomic polynomial. This data representation complicates comparison, requiring costly computations for each bit [5]. There is also a usability concern to consider: a ring such as R_t is not intuitive to most programmers, who expect to operate over 32- or 64-bit integers. While there is a growing interest in supporting more intuitive plaintext spaces within HE, this is still an open research area. We leave addressing this issue for future work.

There are multiple ways to encode integers into the cyclotomic ring R_t, and the choice of encoder has important implications on both which operators are supported and the performance of evaluation. The simplest approach is to encode each integer as a constant polynomial with modulus t chosen to be high enough to support all integer computations without wrapping around mod t. This *naive encoding* is highly inefficient as it typically requires large (multi-precision) values of t, with the ciphertext noise growing as an exponential function of t. A more

efficient *integer encoding* approach is to encode each integer as a binary polynomial and then decode the result of computations by evaluating the polynomial at $x = 2$. This approach can be generalized to rational numbers, which we call the *fractional encoding* [17]. Finally, the *packed encoding* uses the Chinese Remainder Theorem to pack multiple integers into a single polynomial. This allows for SIMD-like operations on encrypted values.

3.2 Julia Programming Language

Julia is an actively developed open-source dynamic high-level programming language designed for computational science, in a similar vein as R or MATLAB. Most relevant to this work, Julia provides powerful reflection and introspection capabilities, as well as a Lisp-like macro system for re-writing the underlying abstract syntax tree. These features have made it easy for us to implement RAMPARTS on top of Julia. For example, `Fhe.jl` is implemented as a standard Julia library without any modifications to the Julia compiler.

4 RAMPARTS "FRONT END" PROCESSING

RAMPARTS provides a Julia library, `Fhe.jl`, that exposes functionality for key generation, encryption, decryption, and evaluation. Table 1 documents (a subset of) the API exposed by `Fhe.jl`. The functions `Fhe.encrypt` and `Fhe.decrypt` provide thin wrappers around the underlying equivalent PALISADE functions that perform encryption and decryption. Thus, we focus this discussion on the workings of `Fhe.evaluate`, where the core of our contribution lies.

Function	Description
`Fhe.keygen(scheme, [f, sizes])`	Generates keypair for HE scheme `scheme`, which is either `BFV` (for the Brakerski/Fan-Vercauteren [11, 18] scheme) or `NULL` (for a "plaintext" scheme for testing purposes). `keygen` optionally takes as input a function `f` and a tuple of sizes, and uses these to automatically parameterize the generated keys.
`Fhe.encrypt(pt)`	Encrypts (plaintext) input `pt`, which can either be an integer or an array of integers, outputting a ciphertext.
`Fhe.decrypt(ct)`	Decrypts ciphertext `ct`.
`Fhe.evaluate(f, args)`	Evaluates Julia function `f` on a tuple of ciphertexts given by `args`.
`@Fhe.evaluate expr`	Macro form of `Fhe.evaluate`; `expr` is an inline Julia expression.
`Fhe.write(fname,ct)`	Serializes a ciphertext `ct` to file `fname`.
`Fhe.write(fname,kt)`	Serializes a key of type `kt`, e.g., secret or public, to file `fname`.
`Fhe.read(fname)`	Deserializes a ciphertext contained in file `fname`.
`Fhe.read(fname,kt)`	Deserializes a key of type `kt`, e.g., secret or public, from file `fname`.

Table 1. A subset of the API exposed by the `Fhe.jl` Julia package.

A programmer causes HE evaluation over a function `f` with input ciphertexts `args` by calling `Fhe.evaluate(f, args)`. Upon execution of `Fhe.evaluate`, we proceed as follows:

(1) We first look up the function `f` and extract the Julia intermediate representation (IR) of the function's code, converting the IR into its "symbolic expression" (s-expression) formulation (cf. §4.1). S-expressions provide a concise way to represent nested data using a Lisp-style syntax.
(2) We next pass the s-expression to CRUCIBLE, our underlying symbolic simulator, which compiles the s-expression into an arithmetic circuit by unrolling loops, removing branches, and removing side-effects (cf. §4.2).

(3) Next, we feed the arithmetic circuit and relevant encrypted inputs to Palisade, which homomorphically evaluates the circuit on the inputs, outputting an encrypted result (cf. §5).
(4) Finally, we map the results to their appropriate Julia type, and also decrypt as required by the program. The result is a Julia value matching the expected type of the output of `f` that can then be used immediately for further processing in the program.

Below, we describe the first two of these steps in more detail. First, however, we briefly describe another use of these steps that makes programming HE in Ramparts easier. When running `Fhe.keygen`, we optionally allow the user to input the function to be evaluated and the expected input sizes. This allows the key generation process to tailor the keys to particular characteristics of the function, namely, by setting appropriate noise growth parameters in the generated keys and also choosing an appropriate plaintext encoding for the computation at hand.

We do this by running Steps (1) and (2) described above, and then analyzing the resulting arithmetic circuit. In particular, we do two main analyses: we calculate the multiplicative depth of the circuit, and we determine appropriate plaintext encoding and plaintext modulus given particular circuit characteristics. As an example of the latter, some HE operations (e.g., native dot product computation) are only supported using particular plaintext encodings (e.g., packed encodings). We can thus search the circuit for particular operators, and choose the appropriate plaintext encoding suitable for those operators. The multiplicative depth, plaintext encoding, and plaintext modulus are then fed to a wrapper of the automated BFV parameter selection capability of Palisade to instantiate a cryptocontext with appropriate parameters.

4.1 Step 1: From f to its S-expression

As discussed in §3.2, the Julia language is designed with meta-programming in mind. Heavily inspired by Lisp, Julia has macro support to allow a programmer to reify and manipulate the abstract syntax tree (AST) of functions written in the language. Ramparts takes advantage of this native meta-programming capability to fetch the AST of `f` *in addition to* the ASTs for all of the Julia functions called *within* `f`.

Our `Fhe.jl` library implements a function, `expand`, that takes `f` and a list of supported primitives (a.k.a., intrinsics), and fetches all the required ASTs from `f` and its called functions *except* for any primitive in the provided list. These intrinsics comprise operators that either we can support directly (e.g., matrix transposition) or that map directly to underlying HE operations (e.g., `+` or `*`). We provide our symbolic simulator with the AST *for all non-primitive functions* used in `f` and Julia's meta-programming facilitates this approach. Additionally, we use s-expressions as an intermediate format. Note that if we were working in a language without meta-programming capabilities, we would need to write a custom parser that fetches the AST of functions from other modules, or repurpose the compiler's (or interpreter's) parser for the language.

4.2 Step 2: From S-expressions to Arithmetic Circuits

In order to generate an arithmetic circuit which is suitable for evaluation under HE, we must remove all loops and branches from the input function. We achieve this by using the Crucible symbolic simulator [2].

At a high-level, symbolic simulation provides a mechanism for generating the *path conditions*[2] of a program. It accomplishes this by having the user specify certain inputs as *symbolic* (i.e., abstract symbols whose values cannot be accessed), and executing the program. Execution for values that are not symbolic proceeds normally, but when a computation or value depends on a symbolic variable, the simulation records the *constraints* that are placed on the symbolic variable. As an example, the statement '`if x > 5 ...`' would place a constraint "symbolic variable `x` must be greater than five" within the branch of the if-statement.

[2] A *path condition* describes a condition that triggers a particular program execution path to be taken.

Our symbolic simulator also effectively simplifies the program as a natural consequence of evaluating it. Compared to traditional compiler optimizations, these simplifications can be thought of as sub-expression elimination, constant folding, and partial evaluation. For example, loops with known bounds are completely unrolled, function calls are expanded out, recursion is removed, and so on.

4.2.1 Why Crucible? Typically symbolic simulators, such as Klee [13], are tailored to a particular language. Crucible, on the other hand, is a language-agnostic system. The trade-off between the two approaches is straightforward: Tools that are language specific require no upfront cost when using their supported language, but cannot be used for languages that do not have identical semantics and memory models as the supported language. Crucible can be a target for arbitrary languages, but requires the user to specify the meaning of types and calling conventions for their language of interest.

Because there is no current symbolic simulation framework for Julia, choosing Crucible freed us from needing to design and implement an entire symbolic simulation engine for Julia. However, we did have the upfront cost of specifying the heap layout of Julia types and specifying Julia's calling convention.

Another trade-off between a custom symbolic simulator and the use of Crucible is in Crucible's use of efficient state merging [24]. Efficient state merging provides a method for symbolic simulators to generate smaller intermediate terms. Because symbolic simulation is a computationally expensive process, this efficiency gain allows Crucible to be exponentially more efficient on certain types of programs, such as programs with bounded loops. This efficiency thus allows Crucible to symbolically simulate larger programs, making Ramparts capable for more complex real-world functions.

Crucible also differs from simulators like Klee [13] in that it must explore all paths through the program. Crucible's symbolic simulation is thus at least as expensive as normal program execution plus the cost of overhead of the symbolic computation, such as computing and storing path conditions. The advantage of exploring all paths is that Crucible can construct a representation of the computation that is faithful to the normal execution of the program for all possible inputs. In turn, this representation is what allows us to compile down to an arithmetic circuit representation.

The downside of needing to explore all program paths is that Crucible has weaker support for programs that use symbolic values in conditionals, such as loop conditions. The weaker support for loops conditions involving symbolic values may seem significant, but in our use case these values *represent ciphertexts in the final circuit*. Currently, no HE library we are aware of supports conditioning on ciphertexts, and thus Crucible's inability to support this does not (currently) affect Ramparts.

4.2.2 Crucible's Use in Ramparts. We take the s-expression produced by the `expand` function described in §4.1 and convert it to Crucible's term representation. This is a straightforward static single assignment form with a set of input variables marked as symbolic. Crucible symbolically executes this input term and the result is a set of output terms that are defined as path conditions over the set of input terms. As long as the input program meets the conditions necessary for processing under our HE system, Crucible can unroll all loops and inline all function calls. The resulting path conditions are in the form of a directed acyclic graph (DAG). We then convert this DAG to an arithmetic circuit[3], which we then pass as input to the backend to evaluate (cf. §5).

Figure 4 shows an example s-expression and arithmetic circuit produced by compiling the Julia function `f(x, y)= x * y + 7`. Our circuit format supports addition, multiplication, subtraction, right shift, and dot product over ciphertexts or plaintext inputs. Each line begins with a wire label, which denotes which wire is associated with the output of the given operator. The `:output` annotation denotes which wire values to output.

[3]The transformation from DAG to arithmetic circuit is only a syntactic transformation, but circuit-specific optimizations can be applied during this phase.

```
(Function f (Array (size 3) (Symbol "#self#") x y) (Expr Void ())
(Expr return (Expr call (GlobalRef (Module Main) (Symbol "+"))
(Expr call (GlobalRef (Module Main) (Symbol "*")) (SlotNumber 2)
(SlotNumber 3)) (Int 7))))
(SymbolicInteger "inp1") (SymbolicInteger "inp2")
(SymbolicExpr (Expr call (GlobalRef (Module Main) f) inp1 inp2))
```

(a) S-expression representation.

```
0 input @ ciphertext
1 input @ ciphertext
2 const @ Integer 7
3 * 0 1
4 + 3 2
:outputs 4
```

(b) Arithmetic circuit representation.

Fig. 4. S-expression (a) and arithmetic circuit (b) representations for Julia function `f(x,y)= x * y + 7`.

4.3 High-level Transformations

One advantage of symbolic simulation is the ability to perform high-level transformations on the code before generating the final circuit. In this section we describe two classes of transformations we explored: specialized gate selection (§4.3.1) and circuit-level optimization (§4.3.2).

4.3.1 Specialized Gate Selection. Certain HE operations may be more efficient for, or only applicable to, particular plaintext encodings. By instructing Crucible to treat these operations as primitive, they can be placed as gates in the final circuit. As a concrete example, we can express matrix multiplication at a high-level using dot products. This in turn gives us the option of replacing dot products with a specialized dot product gate in the final circuit in case the particular plaintext encoding choice exposes a dot product gate that is more efficient than the equivalent sum of products. This functionality gives the HE implementer freedom to experiment with efficient special-purpose gates.

4.3.2 Circuit-Level Optimization. Transforming the Crucible term, or equivalently the circuit representation, can expose optimizations that are inherent in the computation, but not explicit in the program. For example, if the program sums the elements of an array by looping over the array with an accumulator, this results in a circuit that is N-gates deep, where N is the number of elements in the array. This can be rewritten as a tree that is $\log(N) + 1$ deep, by balancing the tree of additions. Of course, this can be accomplished for any associative operation.

Balancing expression trees is particularly beneficial in the case of multiplications. Balanced trees of multiplications not only provide better parallelism but also reduce the multiplicative depth required for circuit computation. We have found this to have significant impact on evaluation running time; see §6.2.1 for an example.

4.4 Other Features of the Ramparts Front End

We now describe two additional features that our choice of both Julia and symbolic simulation exposed: the ability to do "inline HE" (§4.4.1) and program equivalence checking (§4.4.2).

4.4.1 Inline HE Using Julia Macros. As mentioned above, Julia exposes a macro capability that allows one to access and rewrite the abstract syntax tree of Julia code. We use this feature to support inline HE computations. Consider the example in Figure 5. The function on the left depicts a simple iterated matrix computation. Now suppose the programmer wants to outsource the expensive portion of this computation, namely the matrix multiplication, to a remote server. They need only to prepend `@Fhe.evaluate` to the desired HE computation (Line 3); upon execution, this macro rewrites the computation to execute it using HE and returns encrypted result.

4.4.2 Testing and Program Verification. By simplifying the addition of HE to existing Julia programs, Ramparts makes it easy for programmers to test their programs as plaintext computations before switching to HE, which is notably slower in most cases. Our choice of symbolic simulation also opens up the possible use of techniques developed by the program verification community—in particular, program equivalence checking. Given two Crucible terms, we can query an external Satisfiability Modulo Theories (SMT) solver, such as Z3 [4], to determine whether the Crucible terms are equal on all inputs. Program equivalence checking can be used as a way to gain

```
function f(A, B)
  for _ in 1:10
    B = A*B*A'
  end
  B
end
```

```
function f(A, B)
  for _ in 1:10
    B = @Fhe.evaluate A*B*A'
  end
  B
end
```

Fig. 5. Demonstration of inline HE support within `Fhe.jl`. The function on the left operates over plaintext values. The function on the right demonstrates how to use the `@Fhe.evaluate` Julia macro to execute HE evaluation on a single Julia statement (rather than on an entire function).

confidence in an algorithm's correctness by comparing to a reference implementation. We expose such support in `Fhe.jl` through the `Fhe.equiv` function.

We now describe an example use case of how such support came in handy during Ramparts development. One of the example functions we experimented with was the determinant function. Our initial implementation used Laplace expansion with a $O(n!)$ running time for $n \times n$ matrices. Laplace expansion makes for a clear and concise reference implementation for the determinant calculation, but we needed a division-free algorithm with better running time. We found such an algorithm for integer matrices in the literature [8]; after implementing the more efficient determinant algorithm, we used our program equivalence support to show that the two implementations were equivalent on $n \times n$ matrices for $n = 2$ to $n = 8$. This gave us the confidence we needed to use the more efficient implementation.

5 RAMPARTS "BACK END" PROCESSING

We implemented a circuit evaluation interface to the Palisade library to support evaluating circuits output by Crucible. As mentioned in §4.1, we chose to keep the structure and details of the circuit separate from HE configuration details such as scheme parameters, encodings, and the like, so that different backend libraries can be substituted as easily as possible.

The Ramparts backend takes as input an arithmetic circuit file as described in §4.1. Processing begins by parsing the circuit into its corresponding graph representation, which allows for additional (backend-specific) performance analysis, optimizations, and graph restructuring. Large circuits can be composed from interconnected groups of smaller circuits. In schemes where it is necessary to do so, the graph detects depth differences and inserts ModReduce operations (cf. §3.1) at appropriate sites in the graph.

Evaluation proceeds by traversing the graph recursively, one output at a time, until all inputs are visited. Results of evaluations of sub-expressions are cached locally and re-used on subsequent visits to circuit wires. The output of this evaluation is the ciphertexts corresponding to the annotated output wires in the circuit representation.

The circuit itself retains no information about the *types* (e.g., matrices or arrays) of the input or output ciphertexts: the input and output of the circuit are simply value vectors. Thus, when returning output ciphertexts to the Julia user environment, we must map the outputs into the correct output type(s) of the evaluated function. As an example, a user who homomorphically evaluates the sharpening algorithm in Figure 1 expects an output with type `Array{Fhe.Ciphertext,2}`.

For each Julia type used in a Ramparts HE computation, we need two transformations. The first one deconstructs the Julia value into a flat array of ciphertexts. The second one reconstructs the Julia value from a flat array of ciphertexts. For example, for matrices the first transformation flattens the matrix to an array and similarly the second transformation reshapes the array. The first transformation is applied at circuit creation time and is implicit in the structure of the circuit. At the same time, Ramparts writes out a snippet of Julia code that implements the second transformation. Because Julia is a dynamic language, it allows us to easily load and execute this snippet as needed. Currently, Ramparts supports basic array and matrix types; additional

user-defined types must be manually added to Ramparts. In future work, we aim to enhance Ramparts so that Julia programmers can make full use of user-defined types without the need to modify Ramparts.

6 EVALUATION

To evaluate our work, we compared Ramparts to three existing HE systems: Cingulata [1], Palisade [30], and a Python wrapper around Palisade. These three systems represent diverse HE abstraction layers: Cingulata is a compiler-based approach similar in some ways to Ramparts; Palisade represents the lowest level of abstraction consisting of direct API calls to the underlying HE library, and the Python wrapper is a thin abstraction that handles low-level details such as memory management. Our comparison includes programmability and performance of all four systems. For convenience, we list our goals from Section §1 and the sections relevant to each goal:

- G1: Retrofitting can be enabled by abstractions that instantiate HE with few or no hints from programmers or changes to existing code. Section §6.1
- G2: We can leverage those abstractions to automatically transform program constants and variables of diverse data types for interoperability between "in the clear" code and HE. Sections §4.3, §4.4.1, and §5
- G3: High-level language syntax adaptations can be minimized and provided without the need to re-write a compiler. Section §4.1 and §4.4.1
- G4: Symbolic execution of privacy-preserving functions and expressions in Julia programs can generate optimized arithmetic circuits for a rich set of expressions amenable to HE. Section §4.2

6.1 Compared Systems

Cingulata [1] is an open-source toolchain that provides a high-level C++ programming interface for writing HE computations, alongside an optimizer designed to reduce multiplicative depth of circuits evaluated using HE. Cingulata translates algorithms written in C++ into boolean circuits, and then optimizes those circuits using the ABC circuit synthesis suite [12]. As its backend, Cingulata uses the Brakerski/Fan-Vercauteren (BFV) HE scheme [11, 18]. We view Ramparts as similar to Cingulata in that both systems convert programs to circuits, apply optimizations to those circuits, and then execute the circuits using HE. Note that new versions of Cingulata were released after this analysis was performed. The latest version supports the TFHE scheme, which works with Boolean circuits [3], and more advanced circuit optimization/compilation techniques [14, 15].

The Palisade lattice cryptography library [30] implements several HE schemes, and exposes a low-level API for key generation, encryption, evaluation, and decryption. Programming directly in Palisade (as in all existing HE libraries) requires expert knowledge and a deep understanding of the performance tradeoffs to consider when using HE. To simplify this process, Palisade provides a Python wrapper around its C++ API, which directly overloads multiplication, addition, and other operators to instantiate their Palisade HE variants. Both of these approaches directly evaluate the implementation within HE; that is, there is no "compilation" step that builds a circuit representation, and thus no optimizations are applied besides those applied by the programmer directly. Another downside of this approach is that configuring HE parameters becomes a manual process; for example, the programmer must *manually* set the multiplicative depth of computation, whereas with Cingulata and Ramparts this can be done automatically by directly analyzing the compiled circuit.

To examine the differences of programming in these approaches, we show the implementation of the sharpening algorithm for each approach in Figure 6 (the Ramparts implementation can be found in Figure 1).

Looking at the Cingulata code (Figure 6a), we see that it is a straightforward C++ implementation of the sharpening algorithm with two key differences: the `Integer8` type, and ciphertext I/O being handled by `std::cin` and `std::cout`. Unfortunately, these two differences mean that pre-existing C++ algorithms must be re-written to work with HE. Circuit compilation is handled through a call to `FINALIZE_CIRCUIT`, and execution of the circuit (as well as key generation, encryption, and decryption) is handled *outside* the C++ implementation. This non-integrated

```
#include <integer.hxx>
void sharpen(void) {
  Integer8 img[SIZE][SIZE], out[SIZE][SIZE];
  Integer8 weight[3][3] = {{1,1,1}, {1,-8,1},
       {1,1,1}};
  for (int i = 0; i < SIZE; ++i)
    for (int j = 0; j < SIZE; ++j) {
      std::cin >> img[i][j];
      out[i][j] = img[i][j]; }
  for (int x = 1; x < SIZE-2; ++x) {
    for (int y = 1; y < SIZE-2; ++y) {
      Integer8 value = 0;
      for (int i = -1; i <= 1; ++i)
        for (int j = -1; j <= 1; ++j)
          value += weight[i+1][j+1] * img[x+i][y+i
              ];
        out[x][y] = img[x][y] - (value >> 1); }}
  for (int i = 0; i < SIZE; ++i)
    for (int j = 0; j < SIZE; ++j)
      std::cout << out[i][j];
}
int main() {
  sharpen();
  FINALIZE_CIRCUIT("sharpen.blif");
}
```

(a) Sharpening algorithm written in Cingulata.

```
import pycrypto
def evaluate(size, fname):
  crypto = pycrypto.Crypto()
  crypto.DeserializePublicKey("keys", "pk")
  crypto.DeserializeEvalMultKey("keys", "ek")
  dx, dy = int(size), int(size)
  image = []
  for i in range(dx):
    row = []
    for j in range(dy):
      row.append(crypto.DeserializeCiphertext("ct", "%d-%d" %
          (i,j)))
    image.append(row)
  weight = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
  image2 = [[pycrypto.Ciphertext(image[i][j]) for j in range(
      dy)] for i in range(dx)]
  for x in range(1, dx-1):
    for y in range(1, dy-1):
      value = 0
      for i in range(-1, 2):
        for j in range(-1, 2):
          if value == 0:
            value = image[x+i][y+j]*weight[i+1][j+1]
          else:
            value += image[x+i][y+j]*weight[i+1][j+1]
      image2[x][y] = image[x][y] - (value >> 1)
  for i in range(dx):
    for j in range(dy):
      crypto.SerializeCiphertext(image2[i][j], "ct", "%d-%d" %
          (i,j))
```

(b) Sharpening algorithm written in the Palisade python wrapper.

```
using namespace lbcrypto;
void Evaluate(size_t size) {
  CryptoContext<DCRTPoly> cryptoContext = DeserializeContextWithEvalKeys("keys/cryptocontext", "keys/ek");
  int height, width = size, size;
  vector<vector<Ciphertext<DCRTPoly>>> image(height);
  for (int i = 0; i < height; i++) {
        vector<Ciphertext<DCRTPoly>> row(width);
        for (int k = 0; k < width; k++) {
            string ciphertextname = "ct/" + to_string(i+1) + "-" + to_string(k+1);
            Serialized kser;
            SerializableHelper::ReadSerializationFromFile(ciphertextname, &kser);
            Ciphertext<DCRTPoly> ct = cryptoContext->deserializeCiphertext(kser);
            row[k] = ct; }
        image[i] = row; }
  vector<vector<int>> weightsRaw = {{1, 1, 1}, {1, -8, 1}, {1, 1, 1}};
  vector<vector<Plaintext>> weight(weightsRaw.size());
  for (int i = 0; i < (int)weightsRaw.size(); i++)
    for (int k = 0; k < (int)weightsRaw[0].size(); k++)
      weight[i].push_back(cryptoContext->MakeFractionalPlaintext(weightsRaw[i][k]));
  vector<vector<Ciphertext<DCRTPoly>>> image2(image);
  for (int x = 1; x < height-1; x++) {
    for (int y = 1; y < width-1; y++) {
      Ciphertext<DCRTPoly> value;
      for (int i = -1; i < 2; i++)
        for (int j = -1; j < 2; j++)
          if (value == NULL)
            value = cryptoContext->EvalMult(image[x+i][y+j], weight[i+1][j+1]);
          else
            value = cryptoContext->EvalAdd(value,cryptoContext->EvalMult(image[x+i][y+j], weight[i+1][j+1]));
      image2[x][y] = cryptoContext->EvalSub(image[x][y],cryptoContext->EvalRightShift(value, 1)); }}
  for (int i = 0; i < height; i++) {
    for (int k = 0; k < width; k++) {
      string ciphertextname ="ct/" + to_string(i+1) + "-" + to_string(k+1);
      ofstream ctSer(ciphertextname, ios::binary);
      Serialized cSer;
      image2[i][k]->Serialize(&cSer);
      SerializableHelper::WriteSerializationToFile(cSer, ciphertextname); }}
}
```

(c) Sharpening algorithm written in Palisade.

Fig. 6. Code for evaluating the sharpening algorithm using HE written in (a) Cingulata, (b) the Palisade python wrapper, and (c) Palisade. The Cingulata implementation does not contain the (de)serialization code, as this is handled externally. See Figure 1 for the Julia implementation.

environment prevents using the output of an HE computation with other code in the same program, and makes it problematic to introduce HE into an existing application.

The Python implementation (Figure 6b) also looks very similar to a non-HE implementation of the sharpening algorithm. However, consider Lines 20-23. Intuitively, one could remove this if-else statement and instead just retain Line 23; however, this would not work because `value` is initialized as an integer but later used as a ciphertext, and adding an integer to a ciphertext is an unsupported operation. This example shows that adapting a program using the Python wrapper to be HE-compatible may not be straightforward. We also reiterate that the program is executed as written, without the automatic optimizations found in Ramparts and Cingulata. Indeed, it is left to the programmer to be aware of and exploit the subtleties of HE in order to get the best performance. We give an example of where this issue can arise in §6.2. In addition, this approach requires the programmer to manually configure HE parameters, such as multiplicative depth.

Finally, Figure 6c shows the Palisade code, which we can easily see requires the most "HE expertise" to implement: namely, a solid understanding of the Palisade API and knowledge of the tradeoffs of the various plaintext encodings. For example, this approach requires explicit choices on which plaintext encoding to use (see Line 20). These choices are abstracted away in the other approaches (although in the Python wrapper the encoding choice is hardcoded as part of the wrapper logic).

In comparison, the Julia implementation (cf. Figure 1) has *no* HE-specific constructs. It is written in exactly the way a user would write such an algorithm for running on plaintext data in Julia[4]. Our compilation process automatically handles such issues as the different types of `value`, which requires rewriting the logic in the Python implementation. In addition, we automatically select the multiplicative depth by analyzing the compiled circuit—something that neither the Palisade nor Python implementation can do. Homomorphic evaluation and plaintext execution are fully and automatically interoperable within the normal Julia environment. Re-use of existing "plaintext" code in the HE context is also supported, because code need not be re-written to enable HE. Finally, as we show below, these advantages come with little performance impact versus writing directly in the Palisade API.

6.2 Performance Comparison

We ran the sharpening algorithm using all four approaches for images between 8x8 and 96x96 pixels, incrementing by 8 pixels per dimension each time (excluding Cingulata, which we only ran up to 80x80 due to its long running time). We ran all experiments on a standard laptop with an Intel Core i7 CPU @ 2.70 GHz and 16 GB of RAM, using a single core (we thus disabled all parallelism to assure a fair comparison). All systems used the Brakerski/Fan-Vercauteran (BFV) HE scheme [11, 18] with at least 128 bits of security[5]. Both Ramparts and Palisade use the same underlying implementation of BFV [20], whereas Cingulata uses its own implementation; thus, some of the performance differences reported here may be due to implementation differences. In addition, Cingulata differs in that it decomposes all data into single bits, whereas the other approaches directly evaluate the arithmetic representation of the algorithm. This allows Cingulata to directly evaluate, e.g., comparisons, which is not supported by Ramparts[6] or Palisade. However, this flexibility in Cingulata comes at the price of significantly worse performance, as discussed below. For the performance comparison we used Palisade version 1.2.

[4] As mentioned in the introduction, there are some limitations to operators we support, and thus we do not claim that *all* Julia functions are supported. Indeed, we currently support a small subset of the language, and additional features can be added by providing the necessary logic within the symbolic simulator.

[5] In particular, we fixed the root Hermite factor to 1.004.

[6] Note that there is no fundamental difficulty in adapting Ramparts to support boolean operators, and thus comparisons, by decomposing arithmetic operations into their boolean equivalents. This was simply a design choice based on the use of Palisade as the underlying HE library.

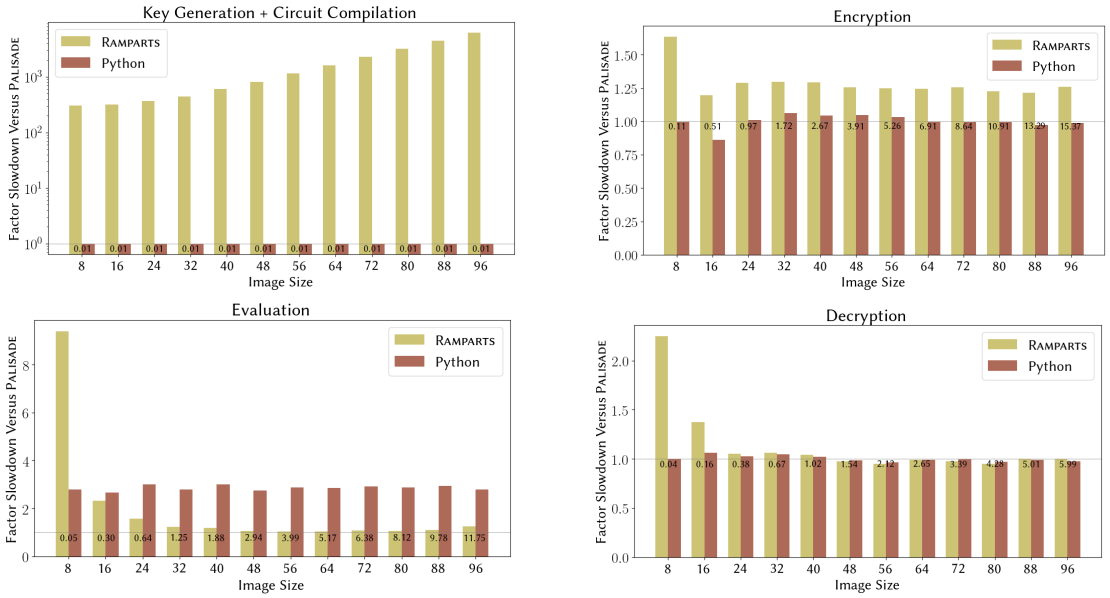

Fig. 7. Factor slowdown of key generation, encryption, evaluation, and decryption, for the 8-neighbor Laplacian sharpening algorithm written in Ramparts and Python, versus the Palisade baseline. Image sizes vary from 8x8 to 96x96, with each bar being the average over 10 runs. The numbers below the gray line denote the running time of Palisade (in seconds). Note that the key generation time for Ramparts is dominated by the time required to compile the circuit; this compilation is not done by either the Python or Palisade implementations.

Figure 7 shows the results of our performance tests. The four graphs in the figure show the performance slowdown of Ramparts and Python versus the handwritten Palisade implementation of the key generation, encryption, evaluation, and decryption phases for the 8-neighbor Laplacian sharpening example from Figure 1. We did not include Cingulata in this comparison due to it being significantly slower than the other approaches; see Figure 8 for a direct comparison between Cingulata and Ramparts.

For completeness, we also include the sizes of serialized ciphertexts and keys, which are the same for Ramparts, Python, and native Palisade implementations as the native JSON-based serialization capability of Palisade v1.2 is used. The sizes of secret and public keys for all image sizes are 41.7 KB and 82.7 KB, respectively. No evaluation keys are needed for the sharpening computation as no homomorphic multiplications or rotations are performed. The size of encrypted image/encrypted result is $s^2 \times 82.7$ KB, where s is the image size. This implies that the communication requirement for the server is $s^2 \times 82.7$ KB (both ingress and egress). Note that all lattice and scheme-specific parameters, which are serialized to a "cryptocontext" JSON object, are stored in the serializations of every ciphertext or key. In this case, the size of this cryptocontext is 766 bytes, which is less than 1% of single ciphertext size (82.7 KB). Hence the communication overhead of duplicating the cryptocontext serialization is practically negligible.

As shown in the graphs, the performance of Ramparts is quite competitive with both Palisade and Python in terms of encryption, evaluation, and decryption; all within a factor of between 1.0× and 1.3×, excluding the 8x8 image, which due to its fast running time (0.05 seconds using Palisade) causes the overhead due to the dynamic nature of Julia to dominate the cost.

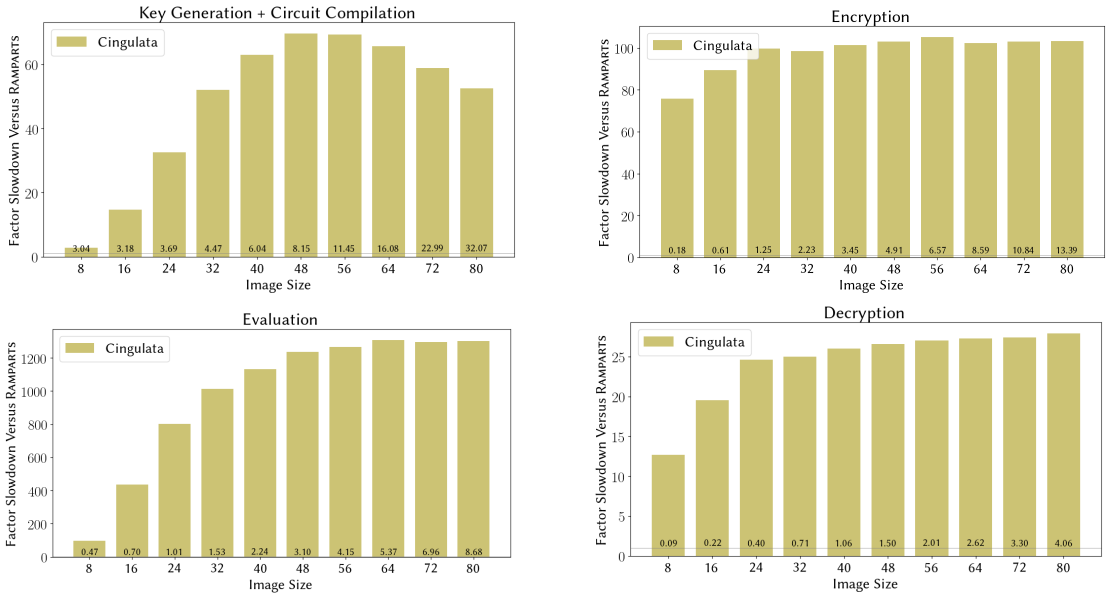

Fig. 8. Factor slowdown of key generation, encryption, evaluation, and decryption, for the 8-neightbor Laplacian sharpening algorithm written in Cingulata, versus the Ramparts baseline. For key generation, this includes the time to compile the circuit for both Cingulata and Ramparts. The numbers above the gray line denote the running time of Ramparts (in seconds). Due to the slow running time, the Cingulata times consist of only a single run.

For key generation we see that Ramparts performs worse (exponential in the size of the image). This is because we compile the circuit as part of key generation in order to automatically parameterize the generated keys with the required multiplicative depth of the computation. However, note that in Palisade and the Python wrapper, the multiplicative depth must be manually computed by the programmer by analyzing the implementation itself. Ramparts alternatively allows the programmer to *manually* set the depth and avoid the additional computation for automatically calculating the depth. Doing so for this example would push the key generation running time to be on par with the Palisade and Python implementations, at the cost of needing to compile the circuit during evaluation. Finally, we note that the compilation of a function on a specific input size needs to happen only *once* in Ramparts; that is, we can re-use compiled circuits across multiple evaluations.

These performance results demonstrate that we do not sacrifice significant performance in terms of HE running time versus directly using Palisade. On the other hand, we gain significantly in both usability as well as flexibility of optimizations, the latter of which we discuss in §6.2.1.

Comparing Ramparts to Cingulata (the only other approach that compiles circuits and automates the calculation of the multiplicative depth of the computation), we find Ramparts significantly more efficient in all four phases of HE, with a factor of 100–1200× speedup in evaluation (cf. Figure 8). We reiterate that we wrote the most basic implementation of the sharpening algorithm in Cingulata and avoided using packed encoding techniques. This, combined with the fact that Cingulata uses boolean circuits versus Ramparts's arithmetic circuits means that readers should cautiously interpret these results. However, the compilation speed is largely unaffected by these choices, and here we can see a significant performance improvement (more than 60× for 48x48 images) using Crucible versus the direct compilation and optimization approach taken by Cingulata.

Finally, we note that we are comparing against a naive Palisade implementation in that the sharpening algorithm can be carefully designed to use the packed encoding based on the Chinese Remainder Theorem, and thus gain significantly in terms of performance due to the SIMD-like parallelism inherent in such an encoding. A Palisade implementation that utilizes the packed encoding takes under 0.1 seconds for all image sizes; however, it requires detailed HE expertise and careful analysis of the algorithm to get it right. A future research direction aims to enhance Ramparts to take advantage of and optimize for the use of packed encodings.

6.2.1 Circuit Optimizations. As discussed in §4.3.2, one benefit of the Ramparts compiler approach is that we can apply optimizations directly to the circuit itself, whereas using an API pushes the use of optimizations onto the programmer. As an example, consider the following simple Julia program for multiplying a list of values:

```
function f(xs)
  result = 1
  for i in 1:length(xs)
    result *= xs[i]
  end
  result
end
```

The naive instantiation of this approach has multiplicative depth equal to the length of `xs` minus one; and this is indeed the behavior we see when using Palisade's Python wrapper. However, flattening this multiplication reduces the multiplicative depth to $\log(|\texttt{xs}|) + 1$—a potentially significant saving. We implemented such an optimization within Ramparts; this brings the running time of evaluating the above function on a length-eight list from 2.5 seconds to 0.5 seconds.

7 RELATED WORK

The problem of building higher-level programming paradigms for secure computation, and homomorphic encryption (HE) in particular, has been addressed in several publications across various secure computation domains, such as secure multi-party computation and verifiable computation.

7.1 Homomorphic Encryption

There has been a large volume of prior work with the general goal of making HE more practical. These prior efforts primarily focused on the design and implementations of more efficient HE libraries, as opposed to higher-level languages for implementing HE. In particular, there has been significant effort in supporting specific HE applications more efficiently [10, 19, 22, 32], allowing non-expert users to apply HE as a point solution—however, these approaches do not allow a non-expert user to develop their own HE applications.

There are several general libraries that support homomorphic encryption, including HElib [21], Palisade [30], SEAL [25], and TFHE [3]. All of these libraries expose implementations of specific HE schemes, but require users to implement application functions at a low level of abstraction using an API. These implementations are effectively too "low-level" for most data analysts to use in practice, as it would take extensive training and experience with HE to use these libraries effectively. These libraries also require the user to build up their function from "first principle" operations, such as addition and multiplication. While components can be abstracted out into separate functions, this approach becomes more and more difficult and error-prone as the complexity of the desired function increases.

Despite the difficulty for non-experts to use these libraries, there has been very little work in making HE more "usable". Aslett et al. [6] implemented the Brakerski/Fan-Vercauteren scheme [11, 18] as a package for the R programming language; however, this still requires manually constructing the circuits from first principles, and thus suffers from many of the same drawbacks as the existing C++ libraries described above. Closer to our work,

Cingulata [1] provides a compiler and run-time environment for homomorphic evaluation of C++ programs. We provide a detailed comparison with Cingulata in §6.

7.2 Secure Multi-Party Computation

Whereas HE has not seen much effort in the development of high-level programming paradigms, the same cannot be said for secure multi-party computation (MPC). Fairplay [27] provided an early approach to secure two-party programming, which was later extended to multi-party scenarios in FairplayMP [7]. More recently, there have been several notable approaches towards supporting higher-level programming for MPC, including Wysteria [31], ObliVM [26], GraphSC [28], Obliv-C [33], and Sharemind [9], to name but a few. These systems provide new programming languages for writing MPC algorithms, with the type system used to enforce security properties in the compiled code.

7.3 Verifiable Computation

Although further afield from HE than MPC, the field of verifiable computation has also seen recent success in developing programming tools to make this technology more usable by non-experts. Example systems include Pinocchio [29] and Geppetto [16], among others. This latter system takes (a subset of) LLVM code as input and, much like Ramparts, uses symbolic simulation to compile the LLVM input into a circuit representation. One disadvantage of using LLVM as the input language is that certain standard compiler optimizations are not applicable to the verifiable computation backend, and there are likely optimizations that are best done at a higher level than LLVM[7].

8 CONCLUSION AND FUTURE WORK

In this work, we present Ramparts, a system for directly evaluating Julia functions over encrypted data. Ramparts provides a simple Julia interface for key generation, encryption, evaluation, and decryption. Both entire functions and "inlined" computations can be executed in an HE context, with inputs directly attained and outputs directly usable within a running Julia program. Automatic optimization and HE parameter selection relieve the programmer from needing to configure or understand the HE execution environment. Performance of Ramparts is competitive with other approaches that do not provide such ease of use. Thus, Ramparts opens the door to making HE usable by non-expert programmers.

What we present here is only a first step towards a larger vision of an HE compiler toolchain and development environment. Moving forward, we would like to move from a circuit-based intermediate language to an "HE assembly language" which would support looping/branching and other common programming constructs. This would both speed up compilation (by removing the need to symbolically simulate the entire function) as well as provide more concise representations of the compiled HE program. This assembly language could then be a target of other frontend compilers, and be used by different backend HE libraries.

ACKNOWLEDGMENTS

This research is based upon work supported in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either express or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

[7]Because Julia itself compiles down to LLVM, early in the development of Ramparts we considered compiling directly from LLVM but concluded that the loss of potentially higher-level HE optimizations would outweigh the generality gained by this approach.

REFERENCES

[1] 2018. Cingulata. (2018). https://github.com/CEA-LIST/Cingulata

[2] 2018. The Crucible Symbolic Simulator. (2018). https://github.com/GaloisInc/crucible

[3] 2018. TFHE: Fast Fully Homomorphic Encryption Library over the Torus. (2018). https://github.com/tfhe/tfhe

[4] 2018. The Z3 Theorem Prover. (2018). https://github.com/Z3Prover/z3

[5] Seiko Arita and Shota Nakasato. 2016. Fully Homomorphic Encryption for Point Numbers. Cryptology ePrint Archive, Report 2016/402. (2016). http://eprint.iacr.org/2016/402.

[6] Louis J. M. Aslett, Pedro M. Esperança, and Chris C. Holmes. 2015. A review of homomorphic encryption and software tools for encrypted statistical machine learning. arXiv preprint 1508.06574. (2015). https://arxiv.org/abs/1508.06574.

[7] Assaf Ben-David, Noam Nisan, and Benny Pinkas. 2008. FairplayMP: a system for secure multi-party computation. In *ACM CCS 08*, Peng Ning, Paul F. Syverson, and Somesh Jha (Eds.). ACM Press, 257–266.

[8] Richard S. Bird. 2011. A simple division-free algorithm for computing determinants. *Inform. Process. Lett.* 111, 21 (2011), 1072–1074.

[9] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS 2008 (LNCS)*, Sushil Jajodia and Javier López (Eds.), Vol. 5283. Springer, Heidelberg, 192–206.

[10] Joppe W. Bos, Kristin Lauter, and Michael Naehrig. 2014. Private predictive analysis on encrypted medical data. *Journal of Biomedical Informatics* 50 (2014), 234–243.

[11] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *CRYPTO 2012 (LNCS)*, Reihaneh Safavi-Naini and Ran Canetti (Eds.), Vol. 7417. Springer, Heidelberg, 868–886.

[12] Robert Brayton and Alan Mishchenko. 2010. ABC: An Academic Industrial-strength Verification Tool. In *CAV*.

[13] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *OSDI*, Vol. 8. 209–224.

[14] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. 2018. A Multi-start Heuristic for Multiplicative Depth Minimization of Boolean Circuits. In *Combinatorial Algorithms*, Ljiljana Brankovic, Joe Ryan, and William F. Smyth (Eds.). Springer International Publishing, Cham, 275–286.

[15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. 2015. Armadillo: A Compilation Chain for Privacy Preserving Applications. In *Proceedings of the 3rd International Workshop on Security in Cloud Computing (SCC '15)*. ACM, New York, NY, USA, 13–19. https://doi.org/10.1145/2732516.2732520

[16] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. 2015. Geppetto: Versatile Verifiable Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 253–270. https://doi.org/10.1109/SP.2015.23

[17] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. 2017. Manual for using homomorphic encryption for bioinformatics. *Proc. IEEE* 105, 3 (2017), 552–567.

[18] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. (2012). http://eprint.iacr.org/2012/144.

[19] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Micahel Naehrig, and John Wernsing. 2016. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*. 201–210.

[20] Shai Halevi, Yuriy Polyakov, and Victor Shoup. 2019. An Improved RNS Variant of the BFV Homomorphic Encryption Scheme. In *Topics in Cryptology – CT-RSA 2019*, Mitsuru Matsui (Ed.). Springer International Publishing, Cham, 83–105.

[21] Shai Halevi and Victor Shoup. 2018. HElib. (2018). https://github.com/shaih/HElib

[22] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. 2018. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*.

[23] Andrey Kim, Yongsoo Song, Miran Kim, Keewoo Lee, and Jung Hee Cheon. 2018. Logistic Regression Model Training based on the Approximate Homomorphic Encryption. Cryptology ePrint Archive, Report 2018/254. (2018). https://eprint.iacr.org/2018/254.

[24] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. 2012. Efficient state merging in symbolic execution. *ACM Sigplan Notices* 47, 6 (2012), 193–204.

[25] Kim Laine, Hao Chen, and Rachel Player. 2018. Simple Encrypted Arithmetic Library. (2018). https://sealcrypto.org

[26] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 359–376. https://doi.org/10.1109/SP.2015.29

[27] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay—A Secure Two-Party Computation System. In *USENIX Security Symposium*, Vol. 4. San Diego, CA, USA, 9.

[28] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 377–394. https://doi.org/10.1109/SP.2015.30

[29] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly Practical Verifiable Computation. In *2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 238–252.

[30] Yuriy Polyakov, Kurt Rohloff, and Gerard W. Ryan. Accessed August 2019. PALISADE Lattice Cryptography Library. (Accessed August 2019). https://palisade-crypto.org/

[31] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. 2014. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 655–670. https://doi.org/10.1109/SP.2014.48

[32] Kurt Rohloff, David B. Cousins, and Daniel Sumorok. 2017. Scalable, Practical VoIP Teleconferencing With End-to-End Homomorphic Encryption. *IEEE Transactions on Information Forensics and Security* 12, 5 (May 2017), 1031–1041.

[33] Samee Zahur and David Evans. 2015. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153. (2015). http://eprint.iacr.org/2015/1153.