HEtest: A Homomorphic Encryption Testing Framework

Mayank Varia[1], Sophia Yakoubov[1], and Yang Yang[2,⋆]

[1] MIT Lincoln Laboratory[⋆⋆], {mayank.varia, sophia.yakoubov}@ll.mit.edu
[2] Blizzard Entertainment, y4n9@alum.mit.edu

Abstract. In this work, we present a generic open-source software framework that can evaluate the correctness and performance of homomorphic encryption software. Our framework, called `HEtest`, automates the entire process of a test: generation of data for testing (such as circuits and inputs), execution of a test, comparison of performance to an insecure baseline, statistical analysis of the test results, and production of a LaTeX report. To illustrate the capability of our framework, we present a case study of our analysis of the open-source HElib homomorphic encryption software. We stress though that `HEtest` is written in a modular fashion, so it can easily be adapted to test any homomorphic encryption software.

1 Introduction

Homomorphic encryption is a cryptographic primitive that enables computation directly on encrypted data. This technology has the potential to change the way that we protect arbitrary computation on the cloud. Moreover, it may be judiciously applied to special-purpose problems such as database searches in order to develop usable technologies [1] with stronger security guarantees than were possible before [4, 18, 20].

The last five years have seen substantial research into the design of homomorphic encryption algorithms [2, 3, 5, 7, 9, 11, 23, 24]. Some of these algorithms have been implemented in software [8, 10, 12, 13]. Evaluations of these software implementations allow the research community to determine the applications that could benefit most from targeted use of homomorphic encryption technology. However, prior evaluations of homomorphic encryption software were tedious to conduct and *ad hoc* in nature. This resulted in evaluations that cannot be directly compared because the data were produced on different platforms. Additionally, while tests were repeatable in principle, the *ad hoc* nature of prior evaluation software made it challenging to reproduce others' test results.

Our main contribution is to provide a generic, open-source framework for the testing of homomorphic encryption schemes. In this paper, we describe the design of our test framework and our use of this framework to evaluate portions of the HElib software package [12, 13].

Homomorphic encryption. The goal of *fully homomorphic encryption* (FHE) research is to design an encryption scheme that is purposely malleable in a specific way to enable computation on encrypted data. More specifically, an FHE scheme has a special operation $\mathsf{Evaluate}$ that takes a circuit representation C of a program and a series of ciphertexts $c_i = \mathsf{Enc}(m_i)$ and returns an encryption of $C(m_1, \ldots, m_k)$.

Due to their number-theoretic properties, many public-key encryption schemes are naturally homomorphic with respect to a single operation like addition or multiplication [6, 17, 19, 22]. An intriguing question, initially posed in 1978 by Rivest et al. [21], is whether there exists an encryption scheme that simultaneously permits the evaluation of *both* addition and multiplication on

[⋆] Work performed while at MIT Lincoln Laboratory.

[⋆⋆] This work is sponsored by the Intelligence Advanced Research Projects Activity under Air Force Contract FA8721-05-C-002. Opinions, interpretations, conclusions and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

ciphertexts. Since these two operations constitute a logically complete set of operations, the vision described above would follow.

In 2009, the seminal work of Gentry [7] affirmatively answered this long-standing question by demonstrating an encryption scheme based on ideal lattices that exploits the ring structure of the ciphertext space to provide Evaluate operations for both addition and multiplication. Gentry split the FHE problem into two components: the design of a *somewhat homomorphic encryption scheme* (SWHE) that permitted a limited number of Evaluate operations and the insight of a *bootstrapping* algorithm that achieved fully homomorphic encryption through the use of multiple applications of SWHE. A slight tweak of this initial scheme was implemented in [8].

After Gentry's initial work, many cryptographers have designed new FHE algorithms [2, 3, 5, 9, 11, 23, 24] that make substantial improvements along several dimensions:

- Improving the performance of a single addition or multiplication Evaluate operation,
- Increasing the number of Evaluate operations possible in a SWHE scheme before bootstrapping,
- Batching multiple plaintext bits into one ciphertext whose bits are operated on in parallel, and
- Basing the cryptography upon weaker, more accepted number-theoretic assumptions.

Some of these improved algorithms have been implemented in software [10, 12, 13]. However, the FHE community lacks a simple benchmarking tool that enables simple comparisons between these homomorphic encryption schemes and with naïve unprotected computation.

Our software. We have designed and developed a software program for evaluating homomorphic encryption schemes called `HEtest` [15, 26]. Our code has been open-sourced for use under a BSD license [16] and is available for download at `https://www.ll.mit.edu/mission/cybersec/softwaretools/hetest/hetest.html`. The `HEtest` software package comprises four components:

1. A *circuit generator* that can create configurable instances of boolean circuits. Customizable options include circuit depth and the desired distribution of gates in a circuit.
2. A *baseline* that parses and evaluates the circuits as quickly as possible without homomorphic encryption, for comparison purposes.
3. A *test harness* that interfaces with any homomorphic encryption software through a communication protocol. The harness executes *test scripts* that repeatedly call for key generation, circuit ingestion (using the circuits generated above), encryption, homomorphic evaluation, and decryption. During a test, the harness captures and logs metrics pertaining to the correctness and performance of the homomorphic encryption software.
4. A *report generator* that (with no human input) analyses the test harness' logs and produces a LaTeX report with tables and graphs that summarize the correctness and performance results (both in absolute terms and relative to the baseline).

The circuit and report generators are written in Python, whereas the baseline and test harness, which are performance-critical, are written in C++.

Our testing. We used our `HEtest` software to evaluate submissions to the Security and Privacy Assurance Research (SPAR) program. The SPAR program was developed in 2010 by the Intelligence Advanced Research Projects Activity (IARPA). Its objective was to design and build new privacy-preserving data searching technologies that are fast and expressive enough to use in practice.

The SPAR program comprised nine research teams who worked on three separate problems, two of which were about the design and implementation of privacy-preserving database or publish-subscribe schemes [4, 20, 14, 25]. The final component of the project focused on the design of homomorphic encryption schemes, with the aim of producing an efficient SWHE building block that could be used in the design of privacy-preserving search algorithms with strong security guarantees. There were two performers involved in the homomorphic encryption component of SPAR: the IBM Thomas J. Watson Research Center (hereafter referred to as IBM) and Stealth Software Technologies, Inc (hereafter referred to as Stealth).

In this paper, we provide a case study for the use of the `HEtest` framework by describing our evaluation of the IBM submission to the SPAR project, which is largely based on the open-source HElib implementation [12, 13]. Thus, the results of our case study can easily be reproduced by interested readers. We wish to emphasize that our discussion of HElib in this paper is merely as a case study: we used `HEtest` to evaluate Stealth's software as well, and all of the code in `HEtest` is written in a modular, extensible fashion, so it can easily be extended to test other homomorphic encryption software with only minor changes to the data generation and baseline code if support is needed for different types of circuits.

Organization. The rest of this paper is organized as follows. In Section 2, we provide a definition for homomorphic encryption and a brief overview of HElib [12, 13]. In Section 3, we describe the process by which `HEtest` generates and stores data. Section 4 describes the test execution framework in `HEtest`. Section 5 explains `HEtest`'s automated data analysis and report generation. Finally, Section 6 provides a case study of the use of our framework to study IBM's HElib software.

2 Overview of Homomorphic Encryption and HElib

A basic public-key encryption scheme has the three algorithms KeyGen, $\mathsf{Encrypt}$, and $\mathsf{Decrypt}$. KeyGen is a randomized algorithm that inputs a security parameter λ and outputs a public/secret key pair (pk, sk). $\mathsf{Encrypt}$ is a randomized algorithm that takes a public key pk and a plaintext message m from the plaintext space $\mathcal{P}$ and outputs a ciphertext c from the ciphertext space $\mathcal{C}$. Finally, $\mathsf{Decrypt}$ is an algorithm that takes as input a secret key sk and a ciphertext c and outputs a plaintext message $m \in \mathcal{P}$. The computational complexity of these algorithms must be polynomial in λ.

A homomorphic encryption scheme has an additional algorithm $\mathsf{Evaluate}$, which takes as input a public key pk, a function represented as a circuit C, and a vector of ciphertexts $\mathbf{c} = (c_1, \ldots, c_w)$ where c_i is the encryption of m_i. It outputs a ciphertext c' that is an encryption under pk of $C(m_1, \ldots, m_w)$, the result of evaluating the circuit C using $m_1, \ldots, m_w$ as inputs. The scheme satisfies the homomorphic property that for all $(pk,\ sk)$, circuits C, and $c_i = \mathsf{Encrypt}(pk, m_i)$,

$$\mathsf{Decrypt}(sk, \mathsf{Evaluate}(pk, C, c_1, \ldots, c_w)) = C(m_1, \ldots, m_w).$$

In recent years, one construction of a homomorphic encryption scheme based on the ring-LWE assumption by Brakerski, Gentry, and Vaikuntanathan [2] has shown promise of becoming "somewhat practical." In this scheme, plaintext bits are represented as coefficients of a polynomial in the ring $\mathbb{F}_p[x]/(f(x))$. Gentry, Halevi, and Smart [11] design a variant of the BGV scheme in which $p = 2$ and $f(x)$ is chosen to be the n-th cyclotomic polynomial $\Phi_n(x)$. IBM's HElib software is an implementation of this cryptosystem, with further optimizations in ciphertext packing or "batching" [24]. Specifically, if the polynomial ring $\Phi_n(x)$ can be factored modulo 2 into ℓ irreducible factors, then there are ℓ "slots" in which one can encode a plaintext bit by application of the Chinese Remainder Theorem for polynomials. Using this construction, addition and multiplication in the polynomial ring $\mathbb{F}_p[x]/(f(x))$ correspond to element-wise addition and multiplication in the vector of slots, giving rise to single instruction multiple data (SIMD) style operations.

If c is the smallest integer such that n divides $p^c - 1$, then $\Phi_n(x)$ factors into $\ell = \phi(n)/c$ irreducible polynomials modulo p, where $\phi(\cdot)$ denotes Euler's totient function. In order to maximize ℓ, one needs to choose an n that minimizes c. However, n is also constrained by the choice of security parameter λ and the maximum circuit depth d.

Gate	Depth (d)
`MULT`	1
`MULTconst`	0.5
`ADD`	0.1
`ADDconst`	0
`SELECT`	0.6
`ROTATE`[3]	0.25 to 0.75

Table 1. Depth of each gate type, as defined by IBM.

In the HElib cryptosystem, the parameter n was set between 4500 and 45000 for $\lambda = 80$ bits of security and $d \in [4, 24]$. These settings yielded batch widths of approximately $\ell \in [256, 1285]$. Their scheme produced ciphertexts with bit-size asymptotically equal to $O(\phi(n) \cdot d \cdot \log(\lambda))$. Since ciphertext blow-up represents an enormous cost of computing data homomorphically, packing multiple independent plaintext bits into a single ciphertext is a substantial improvement to efficiency.

In HElib, the circuit depth d has a special meaning because the various SIMD operations introduce different amounts of noise to the ciphertext. For instance, adding two ciphertexts increases the noise in the resulting ciphertext linearly while multiplying two ciphertexts increases the noise quadratically. Consequently, `MULT` increases the circuit depth substantially more than `ADD`. Table 1 lists the contributions to circuit depth made by the six gate types supported by HElib. We stress that a gate's depth is different than its *level*: the minimum number of gates between it and the input wires.

3 Test Data

In this section, we describe the process of generating circuits and corresponding inputs on which to test homomorphic encryption software such as IBM's HElib or Stealth's software. We stress that our tool can generate a diverse set of circuits using various gate types. In this section, we describe the generation of circuits that IBM supported: deep circuits consisting of arithmetic SIMD gates with fan-in 2. However, we also used `HEtest` to evaluate Stealth's software on wide, shallow Boolean circuits with large fan-in.

We developed a Python script that generated circuit descriptions and inputs based on configurable circuit parameters. Circuit descriptions were output in ASCII-format and stored as text files, later to be parsed by the test harness described in Section 4.1 and the performers' software. In Section 3.1, we discuss the input parameters to our data generation system. In Section 3.2, we discuss the process by which a circuit and a corresponding input are constructed, given those inputs. In Section 3.3, we discuss the format in which the test harness expects the circuit and input data, and in Section 3.4, we discuss the format in which our analysis tools discussed in Section 5 expect the data.

3.1 Generation Parameters

The parameters for circuit generation include desired circuit width w (i.e. number of input wires), circuit depth d, batch size ℓ, the security parameter λ, and the distribution of gate types that the circuit comprises. Optionally, a random seed could be specified to reliably reproduce data. If the seed is omitted, a random one is chosen at runtime. All parameters were contained in a configuration file that the script read.

Most of our tests were run on circuits consisting of a uniformly selected set of gates over the six types shown in Table 1; we refer to these tests as '`mixed`.' However, it was also very useful to be

[3] The depth of a `ROTATE` gate depends on ℓ, the number of plaintexts packed or "batched" into a ciphertext.

able to produce circuits consisting entirely of one of the six gate types, so as to be able to compare the relative efficiency of HElib's evaluation of these gate types. Note that for circuits composed entirely of one gate type, a different notion of depth was needed because some gate types do not contribute to depth d. Therefore, for such single gate type circuits we used the number of levels num_levels in place of depth, referring to the length of the longest path from the output gate to any of the input wires.

3.2 Circuit and Input Generation

Circuits are generated starting with the input wires and ending with the output gate. A total of w input wires were created, and for each subsequent level of the circuit, w gates were created for which either one or two inputs (depending on the gate type) are randomly chosen from amongst the gates and wires of the two levels above it. If the test type was '`mixed`,' generation went on until all of the gates at the last level had depth greater than d; then, a random gate from the set of gates with the correct depth d was chosen to be the output gate. If the test type was not '`mixed`,' generation went on until num_levels levels had been generated, and a random gate from the num_levels$^{\text{th}}$ level was chosen to be the output gate. Once the output gate was chosen, all gates and wires that did not contribute to the output were discarded. For each desired input, w binary strings of length ℓ were generated.

3.3 Test Suite Representation

Once the test data was generated, it had to be stored in a way that was easily accessible to both the test harness and the prototype. Each generated security parameter, circuit, and input was stored in a separate text file. Inputs were naturally represented as lists of binary strings; for instance, here is an example of an input for a circuit with $w = 4$ input wires, each expecting an input of size $\ell = 5$: "[11010,01011,01010,11011]."

Our syntax for describing circuits is somewhat more involved, and we illustrate it here with an example in both written and graphical form in Figure 2. The first line of Figure 2 specifies the number of input wires (w), the depth of the circuit (d) as defined by IBM, and the batch size (ℓ). Following this header, all gates are listed ordered by their level (not depth d). This guarantees that all inputs to a gate are defined before the gate is defined. (Note, however, that gates on the same level appear in arbitrary order.) Each gate is identified by a string containing the character '`G`' followed by a unique id. Note that not all gate ids between 1 and the maximum gate id are represented; this is because not all gates end up contributing to the output gate, and those that do not get dropped. The input wires are indexed by a string containing the character '`W`' followed by a unique id in $0 \ldots w - 1$. Following the gate id, on the same line, is the gate type and a list of the gate's inputs. These can be pointers to input wires, other gates, or constants if the gate type requires a constant input. Note that the wires are not defined separately in the circuit description; they appear only as inputs to gates. Any constants will always be the last of the gate's inputs.

Once the security parameter, circuit and input files were created, a single *test script* file corresponding to the test suite was produced. It contained the paths to the files that stored each data object, in the order in which they were meant to be sent to the performer binaries. This test script facilitates the test execution process. Finally, in order to have a common repository for all test artifacts, the parameters of each circuit and input were stored in a SQLite database that is described in Section 3.4.

```
W=4,D=4,L=5
G4:LMULconst(W2,01101)
G6:LMUL(W2,W0)
G5:LROTATE(W2,4)
G8:LMUL(G6,W2)
G9:LMULconst(G5,00101)
G7:LADD(G4,W0)
G13:LMULconst(G8,00000)
G11:LMULconst(G8,11010)
G14:LSELECT(G11,G7,11001)
G16:LSELECT(G13,G9,00001)
G18:LSELECT(G14,G13,11110)
G19:LADDconst(G16,10000)
G22:LMULconst(G19,01000)
G25:LADD(G22,G18)
G26:LMULconst(G25,11010)
```

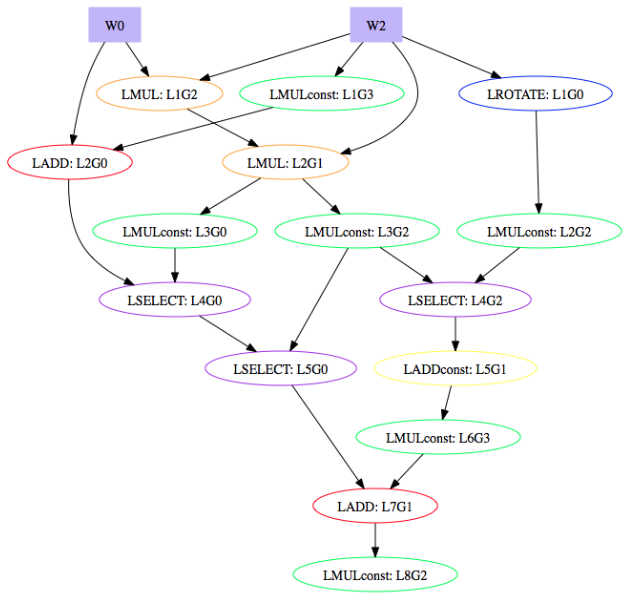


Fig. 2. The syntax of a generated circuit (left) and a graphical illustration of the same circuit (right). In the illustration, gates are labeled first by their level (e.g. ‘L4’) and then by their id within that level (e.g. ‘G2’).

3.4 SQLite Database

`HEtest` uses a SQLite database as a central repository for test information. This database served as the “glue” that enabled integration of the components of our test framework and the automation of our entire test process. We use a SQLite database because it is lightweight, SQL-based, and easy to share and back up.

Our SQLite database is built during the data generation process, and it is initially populated with some descriptive information about the circuits and inputs such as the circuit depth, the number of input wires, and the number of gates of each type present. Baseline and performer test results are later automatically added upon execution of a test, as detailed in Section 4. The SQLite database is used during automatic generation of a report characterizing the correctness and performance of the homomorphic encryption prototype, as described in Section 5. Because of all the circuit- and input-specific information we store, our report can correlate performance with specific circuit parameters, making for a very detailed analysis. We are also able to add new metrics at any time, without having to repeat the entire testing process, because the SQLite database contains all of the necessary information.

4 The Test Framework

The design of our test framework was motivated by three goals. First, we wanted to assess the performance of homomorphic encryption schemes by measuring the duration of key generation, encryption, decryption, and homomorphic evaluation. Second, we wanted to characterize the overhead of privacy assurance by comparing the system that uses homomorphic encryption to one that offers no security. Finally, we wanted to design a test harness that could be used to evaluate arbitrary homomorphic encryption schemes (such as those from IBM and Stealth) in a black-box manner.

The homomorphic encryption software being instrumented by our test framework comprised two processes: a *server* that performed homomorphic evaluation and a *client* that performed key generation, encryption, and decryption. They are collectively called the *system under test*, or SUT.

4.1 The Test Harness

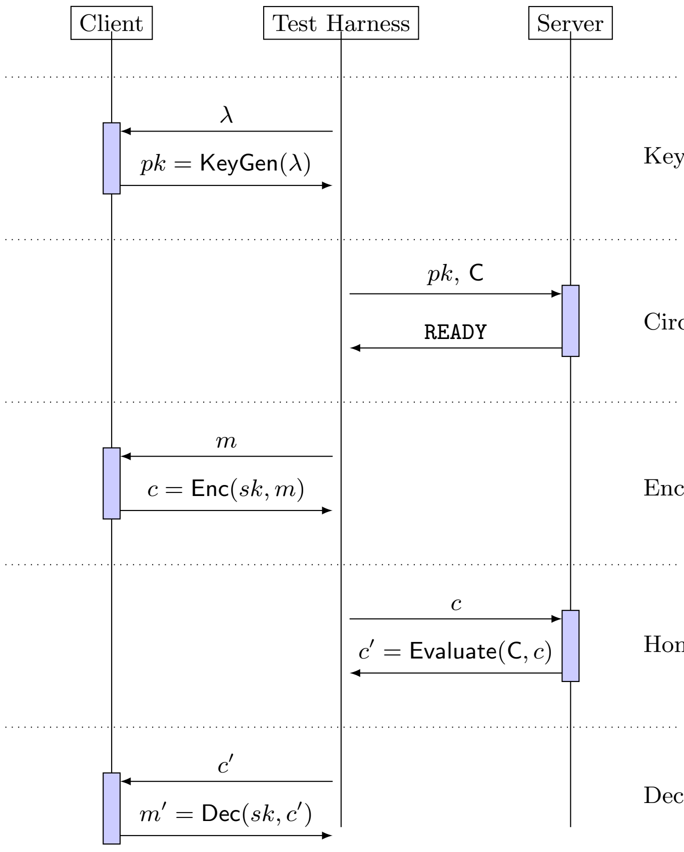
The *test harness*, a program that we developed in C++, spawned the client and server processes of the SUT. It communicated with the SUT through the client's and server's standard input and output streams. After both processes were properly initialized, the test harness called on them to repeatedly perform key generation, encryption, circuit ingestion, homomorphic evaluation, and decryption. A configuration file, read by the test harness on start up, specified the location of files containing the security parameters, plaintext inputs, and circuits to be used.

In the key generation step, the test harness sends the value of the security parameter to the client and receives a public key. In the circuit ingestion step, the public key and circuit description are sent to the server. When the server finishes parsing the circuit description and is ready to accept inputs, it returns a `READY` message to the test harness. In the encryption step, the test harness sends a series of plaintext messages to the client and receives their ciphertexts. Next, in the homomorphic evaluation step, the ciphertexts are sent to the server. The server evaluates the circuit using the ciphertexts as input and returns a ciphertext representing the output to the circuit. Finally, in the decryption step, the test harness forwards the ciphertext returned by the server to the client. The client decrypts the ciphertext and returns the plaintext message. Figure 3 illustrates these steps.

Communication between the test harness and the SUT was dictated by a simple packet-based communication protocol that we developed. A packet consisted of a header, either ASCII-encoded plaintext data or binary-encoded ciphertext data, and a footer. These components were delimited by linefeeds. Security parameters, public keys, plaintext messages, and circuit descriptions were encoded in ASCII while ciphertexts were encoded in binary (with a prefix indicating the size of the binary payload in bytes). Our testing framework assumed that the client and server only communicate through the test harness interface (as shown in Figure 3), never with each other directly.

In each of the steps described above, the duration of the cryptographic operation was measured from when the test harness wrote the first byte of command packet to when last byte of the response packet was read. Consequently, unavoidable communication overhead such as writing data to pipes were captured; however, the latency of these calls were negligible (about 10^{-4} seconds) compared to those of cryptographic operations. Our test harness captured the following metrics:

1. **Evaluation accuracy:** The fraction of circuit evaluations that are correct, i.e. the decrypted result returned by the SUT is equal to the result of directly evaluating the circuit on the plaintext inputs.
2. **Key generation time:** A measure of how long it takes for the client to generate cryptographic keys.
3. **Key size:** The size of public keys generated by the client prototype
4. **Ingestion time:** A measure of how long it takes for the server to prepare a circuit for evaluation.
5. **Encryption time:** A measure of how long it takes for the client to encrypt a plaintext message.
6. **Ciphertext size:** The average size of ciphertext per bit of plaintext input.
7. **Evaluation time:** A measure of how long it takes for the server to evaluate a circuit.
8. **Decryption time:** A measure of how long it takes the client to decrypt a ciphertext.
9. **Total elapsed time:** The sum of the encryption, evaluation, and decryption times.


Client
Test Harness
Server
λ
$pk = \mathsf{KeyGen}(\lambda)$
pk, C
READY
m
$c = \mathsf{Enc}(sk, m)$
c
$c' = \mathsf{Evaluate}(\mathsf{C}, c)$
c'
$m' = \mathsf{Dec}(sk, c')$

	Harness Command	SUT Response
Key Generation	KEY	KEY
	<security parameter>	<public key>
	ENDKEY	ENDKEY
Circuit Ingestion	CIRCUIT	CIRCUIT
	<circuit description>	CIRCUIT READY
	ENDCIRCUIT	ENDCIRCUIT
	KEY	
	<public key>	
	ENDKEY	
Encryption	PDATA	EDATA
	<input1>	<size>
	<input2>	<ciphertext>
	ENDPDATA	ENDEDATA
Hom. Evaluation	EDATA	EDATA
	<size>	<size>
	<ciphertext>	<ciphertext>
	ENDEDATA	ENDEDATA
Decryption	EDATA	PDATA
	<size>	<output>
	<ciphertext>	ENDPDATA
	ENDEDATA	

Fig. 3. Sequence diagram showing data flows during a test (left) and the actual packets sent between the test harness and SUT client/server (right). Angle brackets denote variables to be replaced by actual values.

4.2 The Baseline

In order to characterize performance in a setting without privacy assurance, we provided a *baseline*: a client-server implementation of a circuit evaluator that operates on plaintext inputs. The client offered stubbed implementations of key generation, encryption, and decryption in order to adhere to the communication protocol. They returned properly-formed response packets. The server supported two operations: circuit ingestion and direct evaluation on plaintext inputs.

Like the test harness, we developed the baseline system in C/C++. It was reasonably efficient, with optimizations that would normally be found in circuit evaluation programs. Most notably, it short-circuits gate operations when possible (e.g., an `AND` gate returns false as soon as one of its inputs is determined to be false). In addition, we provided an API for defining new gate types should the need arise.

In the circuit ingestion step, the baseline uses a scanner and parser to read textual circuit descriptions and construct circuits. A scanner, oftentimes called a tokenizer, is a program that recognizes lexical patterns in text. Any circuit description output by our circuit generator tool that we described in Section 3 can be read and tokenized by the scanner. These tokens are subsequently fed into a parser, which constructs circuit gates and eventually builds a circuit.

We recognized that implementing scanners and parsers for circuit descriptions is tedious, error-prone, and non-extensible. As a result, we used two tools, Flex and Lemon, to programmatically generate these software components. Flex is a scanner generator – given a set of rules (i.e. mappings between regular expressions and tokens), it outputs C source code, which when compiled, produces a scanner. The rules that we used to parse IBM-style circuits are found in the `ibm-scanner.l` file in our open-source repository. Lemon is a parser generator – given a context-free grammar, it produces C source code, which when compiled, produces a parser. The grammar for IBM-style circuits is defined in the `ibm-parser.y` file. To extend our baseline to evaluate a new gate type, one would need to add an additional rule, define a new token type, and provide a C++ implementation of the gate that extends the base gate type.

5 Report Generation

After executing a test, the final step in the `HEtest` chain is the production of a concise report that presents the correctness and performance results in such a way that a human can quickly draw intelligent conclusions about the prototype performance.

In the initial version of our software, this process was mostly performed manually with the aid of a few analysis scripts that produced the graphs and tables we desired. However, the addition of any new data necessitated a repetition of the entire process, which was tedious and time-consuming.

To simplify this task, we developed a tool that automatically generated a detailed report describing the performance and correctness of the prototype. In addition to giving us a summary of the system's performance within seconds after the completion of a test, this tool allowed us to identify odd or unexpected behaviors exhibited by the system in near real-time without having to manually search through test data.

The report generator read from a centralized SQLite database containing all of the timing and correctness data, as well as all of the parameters of the tests. It automatically performed analyses of the correctness of the homomorphic encryption software under test (i.e., whether its outputs agree with those from the baseline), and it also determined the dependency of the latency on various factors such as input size, batch size, and circuit depth. It characterized the latency both in absolute terms and relative to the baseline described in Section 4.2.

In Section 6, we display the power of the report generator by showing the results of an execution of `HEtest` on the IBM HElib software. Note that the formats of graphs and tables were stored in easily-updated template files, so the tool could easily be extended to produce a new type of graph or table if desired.

6 Experimental Results

In this section, we present some of the auto-generated analyses that we performed over the HElib test results. All of the statistical analysis and graphs in this section, as well as much of the expository text, were automatically produced by our report generator tool.

We stress that our use of HElib is mainly as a case study. While we do believe that the data about HElib in this section will be of interest to some readers, we are including this data principally to demonstrate the capacity of our `HEtest` tool.

6.1 Experimental Setup

We ran the test harness and performers' system on a Dell PowerEdge R710 server machine with two Intel Xeon X5650 processors and 96 GB of RAM. All software was run on the 64-bit Ubuntu 12.04 LTS Linux distribution.

Values of ℓ	Values of d	Values of w
378	$\{6, 7\}$	{4, 10, 20, 50, 100, 1000}
630	12	{4, 10, 20, 50, 100, 200}
600	18	{4, 10, 20, 50, 100}
682	$\{21, 24\}$	{4, 10, 20, 50, 100}

Table 4. Parameters tested for $k = 80$, based on batch width ℓ, circuit depth d, and maximum number of inputs w.

Throughout this section, all times will be presented in units of seconds and all sizes will be presented in units of bytes; for brevity, we will often omit a statement of units. Also in the interest of brevity, we only present here a subset of our results. For instance, we describe the results for security parameter $k = 80$, which provides 80-bits of security, but omit the results for $k = 128$.

6.2 Real-world Applicability

Before presenting our results, we wish to issue a warning that the data from `HEtest` does not easily translate into intuition about the performance of HElib (or any existing homomorphic encryption scheme) in real-world applications. This is because the optimal representation of the functions in real-world applications is rarely as a circuit; there is almost always conditional logic involved, and no existing homomorphic encryption scheme supports such logic directly. In order to use HElib to securely evaluate a real-world function, the function would first have to be re-written as a circuit, which would almost surely cause a significant slow-down even if it is computed in the clear. This complication motivates the creation of our baseline in Section 4.2: it isolates the slow-down in computing due to homomorphic encryption from that caused by the (inefficient) circuit representation.

Today, HElib can be used for specific small components of real-world applications that can be naturally represented as a circuit; we hope that the analyses in this section will illuminate the costs associated with this.

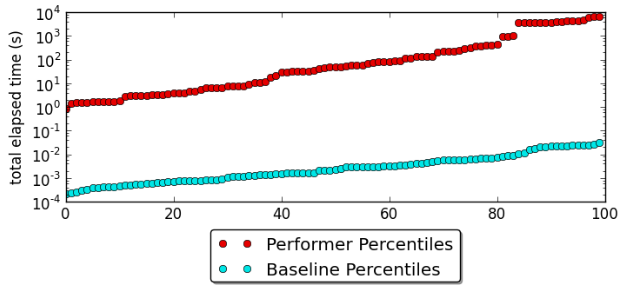
6.3 Parameters Tested

Table 4 describes the parameter values on which we tested HElib. Note that we only include values for $k = 80$; the values of d, ℓ and w are different for $k = 128$. We also tested two additional 'large' circuit settings, with $(\ell = 682, d = 24, w = 200)$ and $(\ell = 1285, d = 60, w = 50)$. For each of the above combinations, we generated two circuits, with five inputs each. Additionally, for each of the six gate types, we generated 20 circuits composed entirely of that gate type, with 5 inputs each. These circuits each had 5 levels, and $w = 100$. Ten circuits of each gate type had $\ell = 6$, and ten had $\ell = 42$.

6.4 Overview of Results

We tested HElib for correctness and performance. Our report generator tool automatically determined that HElib had perfect correctness during the test: for all 1582 circuit/input pairs tested, the outputs from HElib matched those from our baseline. Additionally, the average ratio between the total elapsed time of HElib and an insecure baseline was approximately 52,600.

To provide a more detailed analysis of the total elapsed time for HElib and our insecure baseline, the report generator created the two graphs in Figure 5. The left graph in the figure shows a



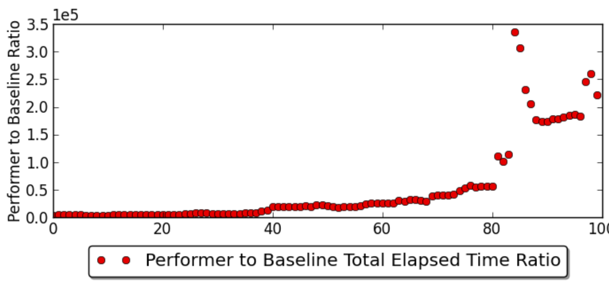


Fig. 5. Total elapsed time percentiles for HElib and the baseline separately (left) and their ratio (right)

histogram of the total elapsed time for HElib and our baseline over the 1582 circuit/input pairs. For legibility, the data are grouped into percentiles: the graph shows the time for the fastest 1% of circuit/input pairs tested (i.e., the 16th fastest test out of 1582), then for the next 1% of tests, and so on. The right graph shows the ratio between HElib and the baseline; in other words, it is the quotient between the two curves on the left graph. These figures demonstrate that the overhead of homomorphic encryption grows as the circuits evaluated become deeper and more complex.

6.5 Key Generation

We found that key generation time and key size were highly correlated with the circuit depth d and batch width ℓ, but were not correlated with the number of inputs w. Note that the combinations of d, ℓ, w, and k were selected jointly for our test at IBM's request, and thus the relationships between the variables may be more complex than presented here.

Figure 9 presents a profile of key generation time varying as a function of circuit depth d and batch width ℓ. In the ranges that we tested, both generation time and key size are linear in ℓ and quadratic in d. Our best-fit model of key generation time is:

$$t = 0.07d^2 - 1.21d + 0.013\ell + 1.4,$$

with an r^2 value of 1.0. Table 6 also provides descriptive statistics for generated key size.

Count	25
Mean	$2.02 \cdot 10^8$
Std Dev	$2.75 \cdot 10^8$
Min	$9.28 \cdot 10^5$
Max	$7.08 \cdot 10^8$

Table 6. Key sizes, in bytes

6.6 Circuit Ingestion

Overall, HElib's circuit ingestion is very fast, and is negligible compared to the time taken for other parts of the scheme. We believe that the ingestion times we observed depended primarily on the

Ingestion	
Count	81
Mean	0.009
Std Dev	0.017
Min	0.0
Max	0.119

Encryption	
Count	999
Mean	0.033
Std Dev	0.304
Min	0.0
Max	5.694

Decryption	
Count	999
Mean	0.116
Std Dev	0.36
Min	0.0
Max	5.091

Table 7. Circuit ingestion latency (left), encryption latency (middle), and decryption latency (right), in seconds

ℓ	Count	Fresh CT size	Evaluated CT size
6	306	5.41 KB	5.65 KB
42	300	203 KB	212 KB
378	121	593 KB	864 KB
600	51	3.60 MB	3.59 MB
630	60	1.66 MB	1.81 MB
682	151	4.16 MB	6.04 MB
1285	10	42.0 MB	46.8 MB

Table 8. Average sizes of fresh and evaluated ciphertexts, as a function of batch size

simple task of parsing rather than on any complexity of the performer's scheme. Circuit description sizes varied in the kilobyte to low megabyte range.

Some basic statistics about circuit ingestion latency are provided in Table 7. Our analysis reveals that ingestion time was mildly correlated with ℓ, but we attribute this to the fact that the bit representation of a circuit description in our format increases with ℓ because gates that have a constant parameter (such as "add a constant to the input" or "multiply by a constant") require ℓ bits to describe.

6.7 Encryption and Decryption

The total encryption time was very fast, with our data (displayed in Table 7) showing that encryption took just 33 milliseconds on average. However, the collected data are contaminated because IBM's software did not adhere to our test harness' communication protocol, so our encryption timer erroneously included network transmission time.

Decryption time is also fast in HElib. While it can take up to 5 seconds for the largest circuits, even this amount of time is negligible compared to the hours such circuits would take for homomorphic evaluation. See Table 7 for detailed statistics.

Finally, `HEtest` captures ciphertext sizes for both "fresh" ciphertexts (i.e., after encryption and before evaluation) and "evaluated" ciphertexts (i.e., after evaluation and before decryption). A summary of these data are shown in Table 8.

6.8 Homomorphic Evaluation

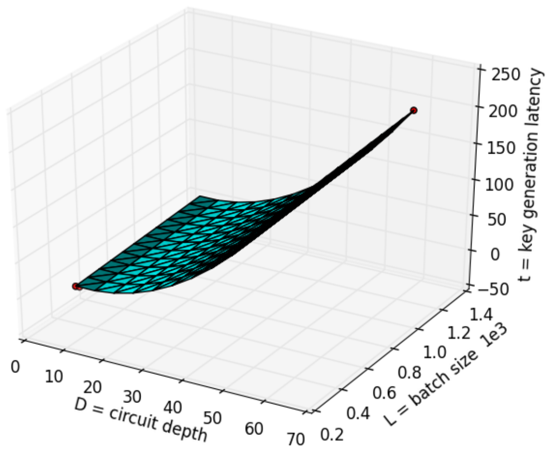
Evaluation time was highly correlated with d and w. It also showed a small correlation with ℓ, but the vast majority of this correlation can be explained as a result of the parameters being selected jointly. The best-fit model for $k = 80$ is

$$t = 2.77d^2 - 74.6d - 1.43w + 0.215wd + 403,$$

with an r^2 value of 0.991. Figure 9 shows a graph of evaluation time.

6.9 Evaluation Time By Gate Type

Finally, we ran several circuits through the performers' system whereby all gates were of the same type. These tests give us an indication of the time required to compute a single gate of each of the various types supported by HElib. Our results are shown in Table 10. Here, our measurements are averaged across all levels of a circuit. If a homomorphic encryption scheme has the property that



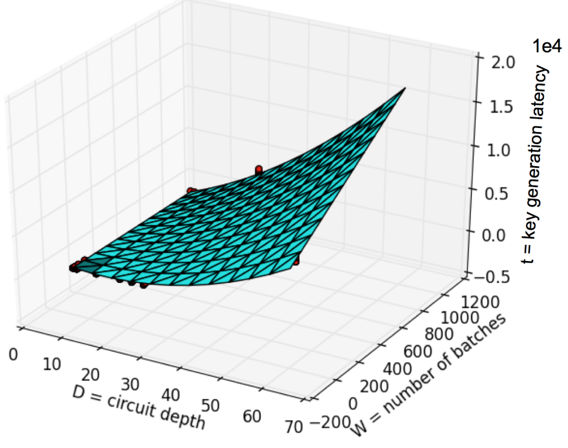


Fig. 9. Key generation time (left) and homomorphic evaluation time (right), in seconds

Gate Type	Count	Mean	Std Dev	Min	Max
`ADD`	101	$2.04 \cdot 10^{-4}$	$1.99 \cdot 10^{-4}$	$1.10 \cdot 10^{-5}$	$6.18 \cdot 10^{-4}$
`ADDconst`	101	$1.85 \cdot 10^{-4}$	$1.75 \cdot 10^{-3}$	$6.00 \cdot 10^{-5}$	$4.43 \cdot 10^{-3}$
`MULT`	101	$1.45 \cdot 10^{-2}$	$1.39 \cdot 10^{-2}$	$4.76 \cdot 10^{-4}$	$3.00 \cdot 10^{-2}$
`MULTconst`	101	$1.92 \cdot 10^{-3}$	$1.82 \cdot 10^{-3}$	$7.60 \cdot 10^{-5}$	$5.19 \cdot 10^{-3}$
`ROTATE`	101	$1.19 \cdot 10^{-2}$	$1.14 \cdot 10^{-2}$	$2.07 \cdot 10^{-4}$	$2.59 \cdot 10^{-2}$
`SELECT`	101	$1.72 \cdot 10^{-3}$	$1.62 \cdot 10^{-3}$	$6.10 \cdot 10^{-5}$	$3.60 \cdot 10^{-3}$

Table 10. Evaluation time per gate, in seconds

the performance of gates various substantially by level, this analysis would not be useful. Due to the special-purpose nature of single gate type tests, note that these data are not included in any of the analyses done in the prior sections.

7 Conclusion

In this work, we built a comprehensive framework for the test and evaluation of homomorphic encryption software with a focus on generalizability, test automation, and integration of test components. We presented a case study application of our `HEtest` software to the IBM HElib software. We stress though that our test framework can be easily adapted to test any other homomorphic encryption software.

We have open-sourced `HEtest` under a BSD license [16]. We encourage interested readers to download our code at `https://www.ll.mit.edu/mission/cybersec/softwaretools/hetest/hetest.html`, and we welcome feedback about our software at `hetest@ll.mit.edu`.

8 Acknowledgements

The authors would like to thank the following people:

– Tim Meunier, for writing the parsers for the test harness output that transfer the test data into the SQLite database.

– Oliver Dain, Nick Hwang and Ben Price, for their help with code reviews and general guidance throughout the software engineering process.
– Mike Depot and John O'Connor, for their IT support during the tests.

References

1. Boneh, D., Gentry, C., Halevi, S., Wang, F., Wu, D.J.: Private database queries using somewhat homomorphic encryption. In: Applied Cryptography and Network Security. vol. 7954, pp. 102–118. Springer (2013)
2. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (Leveled) fully homomorphic encryption without bootstrapping. In: Proceedings of the 3rd Innovations in Theoretical Computer Science Conference. pp. 309–325. ITCS '12, ACM, New York, NY, USA (2012), `http://doi.acm.org/10.1145/2090236.2090262`
3. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: FOCS. pp. 97–106 (2011)
4. Cash, D., Jarecki, S., Jutla, C.S., Krawczyk, H., Rosu, M.C., Steiner, M.: Highly-scalable searchable symmetric encryption with support for boolean queries. In: CRYPTO. LNCS, vol. 8042, pp. 353–373. Springer (2013)
5. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: EUROCRYPT 2010. LNCS, vol. 6110, pp. 24–43. Springer (2010)
6. Gamal, T.E.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory 31(4), 469–472 (1985)
7. Gentry, C.: A fully homomorphic encryption scheme. Ph.D. thesis, Stanford University (2009), `crypto.stanford.edu/craig`
8. Gentry, C., Halevi, S.: Implementing Gentry's fully-homomorphic encryption scheme. In: EUROCRYPT 2011. vol. 6632, pp. 129–148. Springer (2011)
9. Gentry, C., Halevi, S., Smart, N.: Fully homomorphic encryption with polylog overhead. In: EUROCRYPT 2012. LNCS, vol. 7237, pp. 465–482. Springer (2012), full version at `http://eprint.iacr.org/2011/566`
10. Gentry, C., Halevi, S., Smart, N.: Homomorphic evaluation of the AES circuit. In: CRYPTO 2012. LNCS, vol. 7417, pp. 850–867. Springer (2012), full version at `http://eprint.iacr.org/2012/099`
11. Gentry, C., Halevi, S., Smart, N.P.: Better bootstrapping in fully homomorphic encryption. In: Public Key Cryptography - PKC 2012. LNCS, vol. 7293, pp. 1–16. Springer (2012)
12. Halevi, S., Shoup, V.: HElib. `https://github.com/shaih/HElib`, accessed: 2014-09-23
13. Halevi, S., Shoup, V.: Algorithms in HElib. In: CRYPTO 2014. vol. 8616, pp. 554–571. Springer (2014)
14. IARPA: Broad agency announcement IARPA-BAA-11-01: Security and privacy assurance research (SPAR) program. `https://www.fbo.gov/notices/c55e38dbde30cb668f687897d8f01e69` (February 2011)
15. MIT Lincoln Laboratory: HEtest. `https://www.ll.mit.edu/mission/cybersec/softwaretools/hetest/hetest.html` (February 2011)
16. Open Source Initiative: The BSD 2-clause license. `http://opensource.org/licenses/BSD-2-Clause`, accessed: 2014-09-23
17. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EUROCRYPT. pp. 223–238. Springer (1999)
18. Popa, R.A., Redfield, C.M.S., Zeldovich, N., Balakrishnan, H.: CryptDB: Protecting confidentiality with encrypted query processing. In: ACM Symposium on Operating Systems Principles (SOSP 2011) (2011)
19. Rabin, M.O.: Digitalized signatures and public-key functions as intractable as factorization. MIT Laboratory for Computer Science. `http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-212.pdf` (January 1979)
20. Raykova, M., Cui, A., Vo, B., Liu, B., Malkin, T., Bellovin, S.M., Stolfo, S.J.: Usable, secure, private search. IEEE Security & Privacy 10(5), 53–60 (2012)
21. Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation pp. 169–180 (1978)
22. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21(2), 120–126 (1978)
23. Smart, N.P., Vercauteren, F.: Fully homomorphic encryption with relatively small key and ciphertext sizes. In: Public Key Cryptography – PKC 2010. LNCS, vol. 6056, pp. 420–443 (2010)
24. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. In: Designs, Codes and Cryptography. Springer (2011)
25. Varia, M., Price, B., Hwang, N., Cunningham, R., Hamlin, A., Herzog, J., Poland, J., Reschly, M., Yakoubov, S.: Automated assessment of secure search systems. Operating Systems Review (OSR) Special Issue on Repeatability and Sharing of Experimental Artifacts (2015)
26. Yang, Y.: Evaluation of Somewhat Homomorphic Encryption Schemes. Master's thesis, Massachusetts Institute of Technology (2013)