CaPTIF: Comprehensive Performance TestIng Framework

Daniel A. Mayer[1], Orie Steele[1], Susanne Wetzel[1], and Ulrike Meyer[2]

[1] Stevens Institute of Technology, Department of Computer Science, Hoboken, USA
[2] RWTH Aachen University, UMIC Research Center, Aachen, Germany

Abstract

In this paper we present the design and implementation of a framework for comprehensive performance evaluation of algorithms, modules, and libraries. Our framework allows for the definition of well-defined test inputs and the subsequent scheduling and execution of structured tests. In addition, the framework provides a web-based interface for user interaction and allows for the convenient browsing, plotting, and statistical analysis of test results. We furthermore report on our experience in using the new framework in the development of cryptographic protocols and algorithms—specifically in the context of secure multi-party computation.

1 Motivation

When designing practical algorithms, modules, and libraries, experimental performance evaluations are an essential part in determining their suitability for real-world applications. In particular, such performance tests can provide insights which cannot be determined through theoretical analysis alone.

The performance of algorithms in general depends on a variety of choices in the test setup. First of all, more complex algorithms or modules commonly use simpler algorithms as building blocks. When conducting performance tests, assessing the impact of choosing different building blocks providing the same functionality is naturally of great interest. In addition, an algorithm's performance behavior depends on the chosen input as well as many other parameters. When performed manually, varying the input and a larger number of other parameters is time consuming, error prone, and typically generates a large amount of data which needs to be managed and stored. In particular, storing the data such that it may be retrieved together with all the parameters used in a particular test often poses a major challenge in practice. Finally, comprehensive performance tests require an evaluation in a variety of testbeds including different computing platforms and suitable network topologies.

To address these challenges we have developed a test framework which enables structured performance tests of algorithms and modules called *CaPTIF* (*Comprehensive Performance TestIng Framework*). *CaPTIF* is a web-based system which enables its users to define, execute, and review performance tests of algorithms, modules, and libraries. The system keeps track of all parameters used, it schedules the test runs to automatically execute on available test systems, and it provides extensive storage and retrieval functionality for the test results. We have implemented *CaPTIF* and successfully applied it in the development of cryptographic algorithms—specifically in the context of secure multi-party computation [1, 2]. In this paper, we motivate the design of *CaPTIF*, describe its implementation, and share our experience in using the framework.

Outline: We first review related techniques in Section 2. Section 3 discusses the required functionality of a suitable test framework. Section 4 describes our design decisions and Section 5 details the concrete tools we used to implement *CaPTIF*. Finally, Section 6 summarizes the lessons learned when developing and using *CaPTIF*.

2 Related Techniques

In the software development process, performance testing is commonly conducted to determine an application's performance as experienced by the user. In this context, the focus is generally not on a single application session but on the application in its entirety, i.e., one does not test a single, isolated request of a single user but rather the performance when many users interact with the application simultaneously. Software testers commonly use techniques such as load testing, stress testing, etc., to perform these kinds of tests [3]. There is a variety of commercial products and free open-source tools available for the task. For example, IBM's *Rational Performance Tester* [4] and HPs *LoadRunner* [5] both do scalability testing by generating a real work load on the application. Open-source tools such as Apache's *JMeter* [6] and *Grinder* [7] provide a similar functionality.

Motivated by the large cost for commercial performance testing tools, Chen et al. created *Yet Another Performance Testing Framework* [8]. It enables users to create custom test programs which define the business operations to be performed during the test. Chen's framework then executes these tasks concurrently. In [9], Zhang et al. present a cloud-based approach to performance testing of web services. Their system provides a frontend in which users can specify test cases which are then dispatched to Amazon EC2 [10] cloud instances for execution. Similar to all the previous tools, their system is testing the performance under concurrent user access to the system.

The tools described above are geared towards testing of production-stage applications or web-services. In contrast, in this paper the focus is on performance evaluations typically conducted at an earlier stage in the development process. Specifically, we focus on tests which are performed when individual algorithms or protocols are initially designed and implemented. This process typically involves the selection and performance assessment of appropriate building block for the newly designed algorithms or protocols. In addition, these tests are used to not only establish the feasibility of the new design but also to guide the fine-tuning of the implementation. Unlike production-ready testing, the testing in this stage is characterized by assessing and understanding the behavior of an isolated exe-

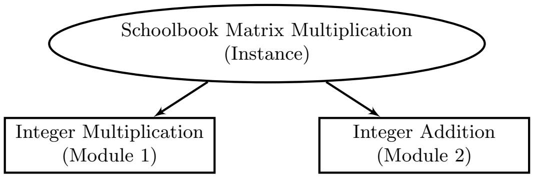


Fig. 1. *Matrix Multiplication* utilizes the modules *Integer Multiplication* and *Integer Addition*.

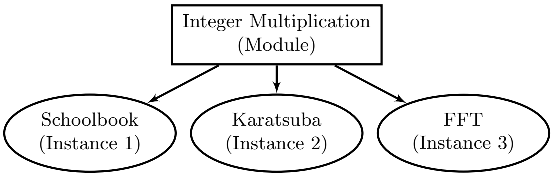


Fig. 2. Different Instances for the module *Integer Multiplication*.

cution of a single protocol or algorithm. This test scenario gives rise to a set of requirements which are not directly supported by the kind of tools mentioned above. In Section 3 we provide a detailed discussion of the requirements for a suitable test framework. To the best of our knowledge, to date there is no publicly available test framework which is focused on such comprehensive and fine-grained performance evaluation of individual protocols and algorithms. It is important to note, that we focus our discussion solely on performance evaluations and leave any kind of correctness tests such as unit tests as an independent problem.

3 Requirements for a Comprehensive Test Framework

In the following, we introduce the requirements for the design of a test framework for comprehensive performance evaluations of algorithms, modules, and libraries.

Below, we motivate that a suitable framework should reflect the modular design of the tested functionality and allow for the precise definition of test inputs. Since comprehensive testing necessitates a separation of the two, they need to be specified independently. In order to conduct a particular test, both can then seamlessly be combined. Furthermore, the framework should provide means for efficient retrieval and analysis of any test results. In the following, we will describe these requirements in greater detail.

Reflecting Modular Algorithm Design: Algorithms often have modular designs. For example, Figure 1 illustrates *Schoolbook Matrix Multiplication* which typically relies on the two *modules Integer Multiplication* and *Integer Addition*. In practice, each of these modules can be instantiated by different concrete algorithms which provide the same functionality, e.g., the module *Integer Multiplication* may be instantiated by the *Schoolbook Method*, *Karatsuba's Algorithm*, or

FFT-based methods (compare Figure 2).[3] In the following, we will refer to the instantiations of a module as *instances*.

As a first requirement, the test framework should reflect the modular structure (indicated in Figures 1 and 2). Specifically, there should be a mapping between the modules and instances in the algorithms and how they are represented in the test framework. In order to provide the means for comprehensively explaining the practical behavior of an instance, the test framework should then allow for flexible testing strategies based on this representation. In particular, a suitable framework should account for the relationships between modules and instances and allow for the testing of different compositions of an instance, i.e., testing for different assignments of instances to modules (e.g., which multiplication method is used to implement integer multiplication in Figure 2).

Test Input: The framework should allow for the definition of well-defined test inputs which are used when conducting a test. In particular, this definition should contain which parameters are used and which parameter(s) is (are) varied within which range. For example, for matrix multiplication, the input consists of two matrices and the parameters are the dimension of the matrices and the size of the individual matrix entries. Test inputs should be defined independently from the tested instance to allow for standardized tests in which test inputs can be re-used in different tests.

Test Execution: A test framework should allow for the execution of tests using a particular combination of an instance and a specific test input. In addition, the framework should record details on the execution environment and on the implementation of the tested instance together with the test. Moreover, in order to enable statistical analysis of the test results, the framework should support the repeated execution of any test.

Flexible Test Environment: The test framework should be flexible in how a particular test is conducted and it should allow for the assignment of a variable number of test systems to a test. For example, the test of a matrix multiplication algorithm can be carried out on a single test system while protocols[4] may need to be executed in a distributed fashion across multiple networked systems. In addition, since mobile devices increasingly gain importance as personal computing platforms, the framework should support different execution platforms such as servers, desktop machines, and mobile devices. For any test utilizing more than one single test system, the network setting has a great influence on for the overall performance and it should be tracked together with the test. For any test environment, a suitable framework should ensure exact performance

[3] Note that this is a simplified example which was chosen to illustrate the underlying concept. In practice, the modular structure of matrix multiplication is not as trivial as described.

[4] A protocol is an algorithm which involves multiple parties. The involved parties perform certain computations locally and exchange messages through some kind of communication channel such as a network.

measurements. In particular, this is challenging in cases where the time for a single execution of an instance is so small that a reliable measurement is not possible due to limitations of the operating system or the hardware.

Implementation Details: Since the performance of an instance may vary greatly depending on the programming language used for its implementation, a test framework should record the implementation details together with a test.

In addition, as the implementation of an instance changes over time and undergoes revisions, a suitable test framework should support efficient re-evaluation of the instance's performance during this process. In order to facilitate a correlation of the changes in the code base with the corresponding performance results, the test framework should allow for the tracking of the respective code revision (e.g., the commits in a version control system) of the instance under evaluation.

Analysis: Comprehensive performance evaluations tend to quickly result in large amounts of test results which need to be organized properly in order to be of any use. The test framework should therefore allow for a structured storing of all test results in combination with the test that produced the specific result. It should be possible to store all test results without prior post-processing in order to allow the user to perform any desired analysis on the raw test results at a later time.

After a test is completed, the test result should be available for review by the user. In particular, the framework should allow the user to efficiently select test results and to display, retrieve, or plot them. In addition, it should be possible to easily compare test results originating from different tests. To assist the user in the analysis of the test results, the framework should provide standard statistical functionality (e.g., computation of averages and error bars based on the standard deviation).

4 Design of the *CaPTIF* Framework

In this section, we introduce the design of *CaPTIF*, our implementation of a test framework which meets the requirements outlined in Section 3. For this, we will refine the description presented in Section 3 and introduce clear terms to define all of *CaPTIF*'s components. Figure 3 shows the main components of *CaPTIF* and illustrates their interdependencies. At its base, *CaPTIF* consists of four components: *Test Configuration*, *Test Case*, *Code Base and Revision*, and *Text Execution Environment*. It is important to note that all four of these components can be specified independently of each other. All four components are combined in the definition of a *Test Run*. Finally, the execution of a test run produces a *Test Result*.

To date, we have applied *CaPTIF* in the area of *secure multi-party computation* (SMPC) [1, 2]. In SMPC, two or more parties wish to compute some function on their private inputs. At the conclusion of an SMPC protocol all parties have only learned the result they are entitled to and, in particular, they

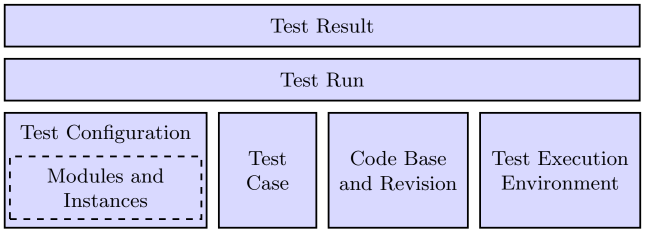


Fig. 3. Main components of *CaPTIF*.

have learned nothing about the other parties' inputs or intermediate results. It is important to note that SMPC operates without the involvement of a trusted third party (see [11] for an introduction to SMPC). Despite this original focus, the final design of *CaPTIF* is far more general and is thus applicable to much more general settings allowing for the effective performance testing of arbitrary algorithms, modules, and libraries. For the sake of clarity of presentation, in this paper we will describe the design of *CaPTIF* on the concrete example of cryptographic protocols in the context of SMPC.

4.1 Mapping Modules and Instances

In Section 3, we have motivated that instances commonly have modular structure. As a first step in our framework design, this section describes how this modular structure is reflected in *CaPTIF*.

Input Types. Since different modules require different types of input, *CaPTIF* allows for the definition of arbitrary *input types* such as integer or set of integers. For each input type it is possible to specify a set P of parameters $p_i \in P$ $(1 \leq i \leq |P|)$ which fully define an input type. For example, a set of integers may be specified by the parameters *set cardinality* and *size of the individual integers in the set*.

Representing the Modular Structure. As discussed in Section 3, any *instance* may rely on different *modules* to implement its functionality and each module can be instantiated by any of the instances corresponding to that module (see Figures 1 and 2). It is important to note that any instance of a particular module may itself rely on other modules in performing its function.

To implement this requirement, *CaPTIF* allows for the specifying of any modules which are used. In this process it is required to indicate which input type (e.g., integer, set of integers) is required to test this module. For each module it is then possible to specify an arbitrary number of instances. In turn, for each instance one can assign all the modules which it relies on as well as any parameters (e.g., *cryptographic key size*) which are required for testing the instance. Overall this yields a complex tree structure which captures the relationships between all modules and instances as illustrated in Figure 4 on the

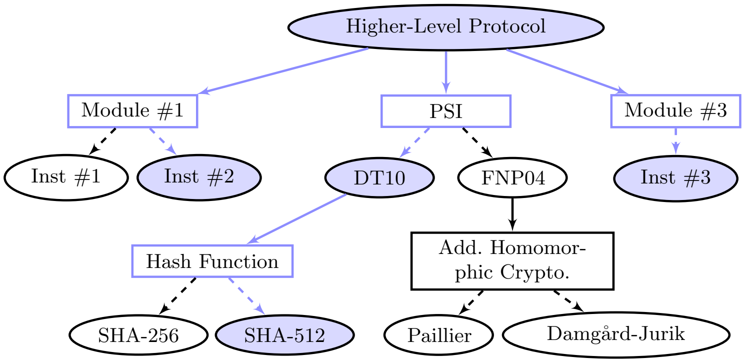


Fig. 4. Illustration of a higher-level protocol and the modules (rectangles) it utilizes. The instances (ovals) shaded in blue represent one possible configuration of the higher-level protocol.

example of a fictitious cryptographic protocol. Here, a higher-level protocol instance is composed of three modules one of which is *Private Set Intersection* (PSI)[5] [12] (all others are left unspecified). The PSI module can be instantiated using any of the various PSI protocols proposed in the literature, e.g., *DT10* [13] or *FNP04* [12]. In turn, *DT10* uses a cryptographic hash functions and *FNP04* uses an additively homomorphic cryptosystem[6] ([15, 16]) as module. Again, different instances for each of these modules exist.

Given this tree, it is possible to perform tests of any subtree rooted at an instance. For example, in Figure 4 one may test the entire high-level protocol, the implementation of DT10, the Paillier cryptosystem [15], or any other instance.

4.2 Test Configurations

Given a modular instance such as the one illustrated in Figure 4, one needs to clearly define its composition, i.e., the assignment of one instance to each module, that is to be evaluated (see Section 3). In the following we refer to this as *test configuration.*

In order to define a test configuration, *CaPTIF* first allows the choosing of the instance which is tested. Next, it is possible to assign one instance to each module within the subtree rooted at the tested instance. For this, *CaPTIF* enables the recursive selecting of one instance for each defined module until leaf nodes are reached. As an example, one possible test configuration for the Higher-Level Protocol is marked by the blue shading applied in Figure 4. Here, Instance

[5] PSI is a prominent SMPC protocol which has applications in a variety of contexts. In PSI, two parties each hold a private input set and at the conclusion of the protocol one party learns which set elements both have in common and the other party learns nothing.

[6] Informally, an additively homomorphic cryptosystem allows for the computation of the sum of two values solely by performing an operation on their respective ciphertexts (without knowledge of the secret key). See, e.g., [14] for an introduction.

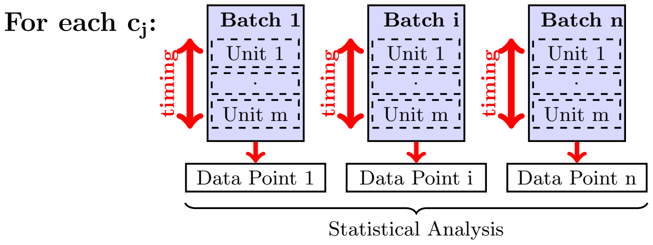


Fig. 5. The structure of the test input for each c_j.

#2 is selected for Module #1, the DT10 PSI using the SHA-512 hash function for PSI, and Instance #3 for Module #3. To reiterate (see Section 4.1), *CaPTIF* does not require to start the test configuration with the root, i.e., Higher-Level Protocol in Figure 4. Instead one can test any subtree, e.g., in order to test DT10 PSI, one would start at the corresponding node and one would only specify the hash function used.

4.3 Test Cases

As established in Section 3, it is crucial to precisely define the used test input. *CaPTIF* implements this by means of a so-called *test case*[7] which is a concise descriptions of how to generate a specific test input.

To specify a test case, *CaPTIF* first requires the selection of the input type (e.g., integer, set of integers). Note that a test case can later only be used in conjunction with a test configuration which requires a test case of the same input type (see Sections 4.1 and 4.2). As motivated in Section 4.1, the input type defines the set P of all parameters $p_i \in P$ which need to be specified in order to fully define a test case of the given input type. For all parameters p_i it is possible to either specify the range (and step size) in which they are varied or set a constant value which is used for the entire test case. As an example, consider an input of type set of integers with p_1 being the size of the individual set elements and p_2 being the set cardinality. One can then choose to vary p_1 from 32 bits to 64 bits in steps of 32 bits and vary p_2 from 10 to 30 in steps of 10.

A second component of the test case is to specify so called *test units* and *test batches*—which are illustrated in Figure 5. A test unit represents a part of the test input which is used for a single execution of the tested instance. In cases where a single execution cannot be measured reliably (because the execution time is below the resolution of the timing function of the operating system), m test units are grouped together into a test batch. The execution for the tested instance is then measured and stored for the entire batch, i.e., for all m executions

[7] This is not to be confused with the notion of a test case used in software correctness testing [17].

in total, and thus the test result contains one data point for each batch.[8] When defining a test case, *CaPTIF* allows for the specification of the number m of test units within each batch. In addition, to allow for the computation of statistics, one can specify the desired number of batches n (see Figure 5).

Once a test case is defined in the test framework, the user can choose to automatically generate and store the corresponding test input by means of test generation scripts which are implemented as part of *CaPTIF* (see Section 5.2). The test input encompasses all the input data that is generated from a given test case.[9] For this, *CaPTIF* first converts the specified parameter ranges into sets V_{p_i} which contain all values in the given range for parameter $p_i \in P$. Subsequently, the Cartesian product $C = \prod_{p_i \in P} V_{p_i}$ of all values is computed. As a result, each tuple $c_j \in C$ $(1 \leq j \leq |C|)$ is a unique combination of parameter values. For example, let $P = \{p_1, p_2\}$ where p_1 and p_2 are varied as above. Then $V_{p_1} = \{32, 64\}$, $V_{p_2} = \{10, 20, 30\}$, and $C = V_{p_1} \times V_{p_2} = \{(32, 10), (64, 10), (32, 20), (64, 20), (32, 30), (64, 30)\}$ with $c_3 = (32, 20)$. For each c_j, n batches containing m units of test input each are generated. Consequently, the overall test input consists of $|C| \cdot m \cdot n$ test units and the corresponding test result will contain a total of $n \cdot |C|$ data points. In the example above, n batches with m test units each would be generated for each pair $c_j \in \{(32, 10), (64, 10), (32, 20), (64, 20), (32, 30), (64, 30)\}$. Once the test input is generated, it is stored and used in all test runs involving this test case.[10]

4.4 Code Bases and Test Environments

Code Bases: *CaPTIF* allows for the definition of different code bases which capture different implementations or programming languages, e.g., *C++* or *Java*. For each test run it is possible to specify which code base and which code revision is used.

Test Environments: *CaPTIF* maintains information on the available test systems and their locations. To date, *CaPTIF*'s design refrains from defining the concrete network topology or latency between each pair of locations since this information grows quadratically in the number of locations and is difficult to maintain. Instead, keeping only the information on the location itself allows for a very good estimate of the respective topology. Moreover, one could obtain exact latency information by performing external tests on the connection between these two location. While this approach only provides for a rough estimate on the topology and latencies, we believe it is sufficient for modeling most practical scenarios.

[8] It is important to note that the total timing for all m execution is stored as part of the test result. In particular, we do not average over the m units i.e., we do not divide the measured timing by m.

[9] It is important to note that the test input for a multi-party protocol includes individual, yet correlated inputs for each party.

[10] It is important to note that any test input that is not automatically generated by *CaPTIF* has to be generated in a similar manner elsewhere.

4.5 Test Runs

To compose a test run, it is necessary to select the desired test configuration and the test case. In addition, an appropriate number of test systems is assigned to the test run and the underlying code base and revision must be specified.

Instead of only allowing for the collection of one time measurement per test batch, *CaPTIF* provides for flexibility by allowing for the collection of multiple *split times*. For example, when testing a cryptosystem, one may time encryption and decryption individually for each test batch.

5 Implementation

CaPTIF stores the details on all modules and instances, test configurations, test cases, test inputs, and test results for all test runs in a relational database backend (see Section 5.1). In order to interact with the database, *CaPTIF* provides a web-based front-end. In this section, we present our choices for appropriate software frameworks, libraries, tools and the concrete implementation details. One crucial step was the selection of the development framework for the implementation of *CaPTIF*. Since *CaPTIF*'s data is not only accessed by the user but also by programs and scripts, e.g., during the generation of test inputs (see Section 5.2), the web interface should be separated logically from the data and application logic. Thus, the *Model-Viewer-Controller* (MVC) pattern [18] which supports all of these requirements was chosen as a basis for implementing *CaPTIF*. The decision for this design pattern was further supported by the fact that MVC frameworks became state-of-the-art in web application development in recent years.

Pyramid [19] is a Python MVC web framework which achieves modularity through extensive usage of the *Web Server Gateway Interface* (WSGI). This modularity and the resulting support for a wide range of different components, e.g., database abstraction and templating make this framework very flexible, light-weight, and thus well-suited for implementing *CaPTIF*. The current version of *CaPTIF* is built using pyramid in conjunction with *SQLAlchemy* [20] as *Object-Relational-Mapper* and *Mako* [21] as template engine for the web interface.

5.1 Backend Database

As stated above, *CaPTIF*'s entire data is stored in a relational database. We use the performance-oriented *MySQL* database using the *innoDB* storage engine which enforces foreign key relationships and ensures data integrity, which is crucial for maintaining all the complex relationships between *CaPTIF*'s components [22]. The entity relationship diagram for the database was carefully designed to support the efficient storage and retrieval of all required data together with its relationships and dependencies.

5.2 Test Input Generation

When defining a test case in *CaPTIF*, one can choose to automatically generate the test input based on the description given in the test case. To facilitate test input generation, an automated Python script regularly checks whether there is any test case for which no test input was generated yet. If this is the case, the script retrieves all information concerning this test case from the database. Based on the input type specified for the test case, the actual test input generation is handed off to an appropriate generator script. *CaPTIF*'s functionality can be extended by adding new generator scripts as required for any new input types that may be defined in the future. Since *CaPTIF* handles the entire interaction with the database, the input generators only need to implement the actual generation of the input.[11] The resulting test input is then stored in the backend database. For efficiency reasons, the entire test input per batch (see Figure 5) is stored as a database blob. This eliminates a great number of database joins which otherwise would cause a significant overhead when the test input is accessed. Furthermore, it makes the database schema independent of the underlying format of the test input. In particular, we store the test input in *JavaScript Object Notation* (JSON) [23] and apply a hexadecimal encoding to any integer value. In addition, since the *CaPTIF* API described in Section 5.3 utilizes JSON to encode requests and responses, storing test inputs in JSON format eliminates costly data conversion during input retrieval. Once the generation is completed, the input is marked as being available and a test run using this test input may start executing.

5.3 Test Execution

In order to facilitate the execution of test runs on different test systems, *CaPTIF* provides an *Application Programming Interface* (API). The use of this API is two-fold. First, it allows the retrieval of a test run from the database. Second, once the test run is completed, the raw test result can be pushed back into the framework where it is associated with the corresponding test run. This separation enables the execution of test runs on devices which can not communicate directly with *CaPTIF* . The latter is of particular importance when testing on mobile devices. Figure 6 outlines the workflow for the execution of a test run. *CaPTIF*'s API is implemented using HTTPS requests in JSON [23] and currently three API calls are supported:

`getTestRun`**:** Check whether a test run was scheduled to run on a particular test system.

`getTestInput`**:** Retrieve the test input associated with a specific test run and test system.

[11] Test generators may internally use a source of randomness to generate randomized input for the tests. Currently, the seed for any random number generator is not stored with the test input.

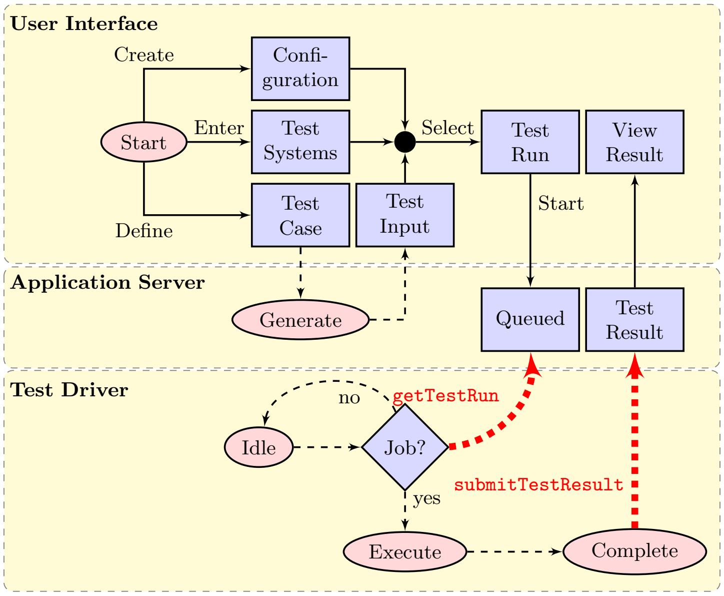


Fig. 6. Workflow when creating and executing a test run using *CaPTIF*. The figure shows the steps taken in the web interface (top), by the application server (middle), and by the test driver executing on the test systems (bottom). Dotted lines indicate automated events while solid lines are user actions.

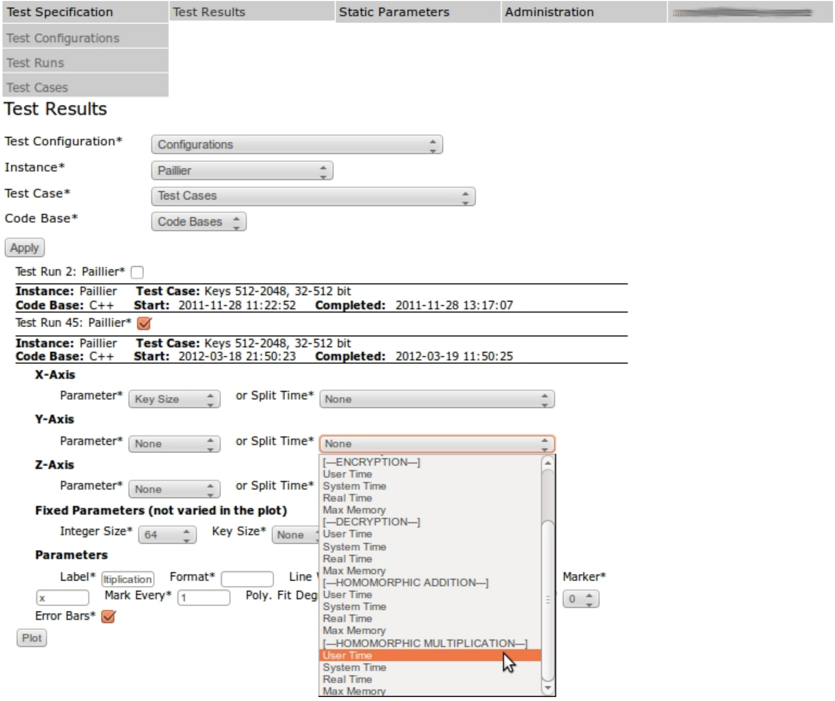
`submitTestResult`**:** Submit the test result from an individual test system to *CaPTIF*.

Each data point in the test result includes all split times obtained for a test batch. Currently, each split time consists of the *user time*, *system time*, *real time*, and *maximum memory used*. However, additional metrics such as, e.g., throughput could easily be added. Below we show *CaPTIF*'s flexibility in designing the execution setting on the example of a test execution on a Linux host and on a mobile device.

Execution on a Linux Host: To execute a test, each test system periodically executes a Python test driver which calls `getTestRun` to check whether a new test run has been scheduled. If a new test run is found, it downloads the associated test input by calling `getTestInput` and executes an appropriate test program. Note that it is not possible to use a generic test program for all types of modules. This is due to the fact that the test requirements vary based on the module that is being tested. Therefore, the test driver calls a different test program[12] which can be specified in the web interface when creating the test run.

After calling the test program, the driver passes the test input via `STDIN` and waits for the test program to terminate. After completion of the test run,

[12] The test program is not part of *CaPTIF* but it is created and compiled by the user ahead of time to facilitate the testing of a specific algorithm or protocol.



Test Specification | Test Results | Static Parameters | Administration | [redacted]

Test Configurations
Test Runs
Test Cases

Test Results

Test Configuration* Configurations
Instance* Paillier
Test Case* Test Cases
Code Base* Code Bases

Apply

Test Run 2: Paillier* □

Instance: Paillier **Test Case:** Keys 512-2048, 32-512 bit
Code Base: C++ **Start:** 2011-11-28 11:22:52 **Completed:** 2011-11-28 13:17:07

Test Run 45: Paillier* ☑

Instance: Paillier **Test Case:** Keys 512-2048, 32-512 bit
Code Base: C++ **Start:** 2012-03-18 21:50:23 **Completed:** 2012-03-19 11:50:25

X-Axis
Parameter* Key Size or Split Time* None

Y-Axis
Parameter* None or Split Time* None

Z-Axis
Parameter* None or Split Time*

Fixed Parameters (not varied in the plot)
Integer Size* 64 Key Size* None

Parameters
Label* ltiplication Format* Line Marker* 0
x Mark Every* 1 Poly. Fit Deg
Error Bars* ☑

Plot

Fig. 7. Screenshot of our web interface illustrating how the plot in Figure 8 was created. The result from testing the Paillier cryptosystem [15] were selected for plotting. The plot assigns the *key size* to the x-axis, the *user time* for the homomorphic scalar multiplication to the y-axis, and fixes the integer size to 64 bits.

the driver collects the test result from `STDOUT` and submits them to *CaPTIF* via `submitTestResults`. This submission also marks the test run as completed within the test database and triggers an email to the user who created the test run notifying her of the completed test.

Execution on a Mobile Device: If the test environment allows it, a mobile device may use a procedure similar to the one used for Linux hosts. However, in settings where this is not possible, a test driver can be executed on a desktop computer to retrieve the test input and test configuration for the device and store them in a temporary file. This file can then be transferred to the mobile device where the user manually executes the test program using the test input copied over from the desktop computer. Similarly, the test result is written to a file on the device, copied back to the desktop computer, and submitted to *CaPTIF*.

5.4 Result Browser

The web interface provides a result browser which can be used to review the test results. The browser displays a list of test runs which can be filtered by test configuration, test instance, test case, etc. *CaPTIF* allows that any result

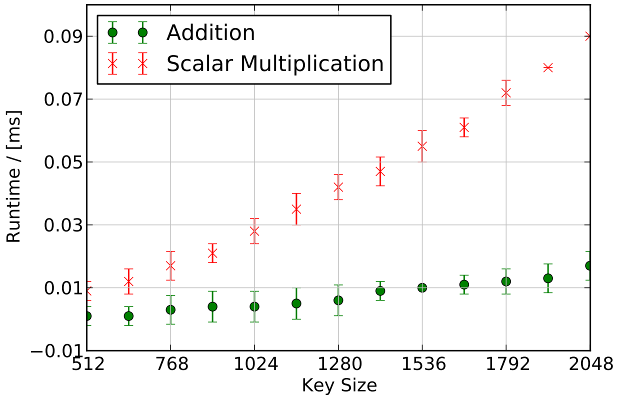


Fig. 8. Example plot generated using *CaPTIF* showing the runtime and corresponding error bars for the homomorphic operations of the Paillier cryptosystem [15].

data can directly be plotted in the browser which enables a quick comparison of test results from different test runs without time-consuming data export/import, plotting, and formatting. When a test run is selected, the corresponding plotting options are displayed using JavaScript (see Figure 7). For each axis of the plot, the user is presented with various choices for values to be plotted. One can either choose one of the parameters defined in the test case or any of the collected metrics (such as *user time*) for any of the recorded split times. If more than one parameter was varied, the remaining free parameters have to be fixed in order for the plot to be meaningful. For example, if an encryption function was tested by varying both the key size and the size of the plaintext, plotting the *user time* as a function of the plaintext size requires the specification of a fixed key size. Furthermore, if the test involved more than one party or host, the user can choose the party for which the test result is to be plotted.

In addition, one can specify a label to be displayed in the legend, the format of the plot (color, width, markers, etc.), and whether error bars should be plotted. One can also choose to fit the test result by a polynomial function. Once all details are specified, the plot is added to the list of current plots. Our implementation in *CaPTIF* leverages the Python *matplotlib* [24] to generate publication quality plots. Multiple test results can be selected for plotting and viewing in the browser. The user has the option to download the resulting figure as *Portable Network Graphics* PNG image or as *Portable Document Format* PDF. All plots created by *CaPTIF* show an average taken over all batches and units. In addition, one may download the raw timings in CSV format which enables more complex analysis using any software of choice. In addition, the web interface provides various options to customize the plot, such as the range of the axes, axes labels, figure dimensions, and a scaling factor which is useful for unit conversions (e.g., milliseconds to seconds). Figure 8 shows an example of a plot generated using *CaPTIF*.

6 Lessons Learned and Future Work

We have successfully used *CaPTIF* to conduct extensive performance evaluations of various cryptographic algorithms (e.g., cryptosystems and other cryptographic primitives), and complex multi-party protocols. These tests involved dedicated servers connected via Ethernet as well as mobile devices connected via Bluetooth and Wi-Fi. While our experience is by and large qualitative in nature and is mostly related to cryptographic settings, we believe that the lessons learned are directly transferable to the testing of general algorithms and protocols.

Before using *CaPTIF*, organizing all the parameters and details of a test run and combining them with the corresponding test results proved challenging. Our experience with *CaPTIF* shows that storing all information in a structured fashion ensures that no information is lost and that it is easily accessible and searchable at any given time. In addition, having all test results stored homogeneously in one location enables fast comparison and plotting. This centralized data storage is particularly useful to us since we are working in a team with members working from different locations.

In addition, the ability of conveniently re-using a test case to evaluate another instance or the same instance using different parameters significantly simplified testing. In particular, the obtained results can directly be compared in a meaningful manner. As part of future work we plan to extend *CaPTIF* to store additional details on test runs such as the versions of any libraries used as well as details on compiler flags, etc.

Finally, test runs can efficiently be scheduled within *CaPTIF* and they are then executed without requiring user interaction. As a result, it is no longer necessary to manually log into each test system in order to start a test run. Moreover, once a test run is completed, the next one is automatically executed which increases the utilization of the test systems and reduces turn-around time. It would be helpful if *CaPTIF* would allow for the storing of more details on the underlying network topology—one requirement discussed in Section 3 but not yet implemented in *CaPTIF*. In addition, it might be desirable to further automate the scheduling process such that test systems are assigned automatically to queued test runs. This would not only reduce the effort required to create a new test run, but it would also ensure that a test run does not wait for a specific test system while others are available. Moreover, this would allow for the execution of a single test run to be split across multiple identical test systems. Furthermore, the test driver could be extended to support the automated fetching of the required code from a repository and the building of the tested binary.

7 Acknowledgements

The authors would like to thank Georg Neugebauer for useful discussions during the design phase of *CaPTIF*. In part, this work was supported by NSF Award CCF 1018616, the DFG, and by the UMIC Research Center, RWTH Aachen University.

References

1. Yao, A.: Protocols for Secure Computations. In: Foundations of Computer Science. Volume 23., IEEE (1982) 160–164
2. Goldreich, O.: Foundations of Cryptography: Volume 2, Basic Applications. Volume 2. Cambridge University Press (2009)
3. Molyneaux, I.: The Art of Application Performance Testing. Volume 1. O'Reilly Media (2009)
4. IBM: Rational Performance Tester. http://www-01.ibm.com/software/awdtools/tester/performance/
5. Hewlett Packard: HP LoadRunner. http://www8.hp.com/us/en/software/software-product.html?compURI=tcm:245-935779
6. Apache Software Foundation: Apache JMeter. http://jmeter.apache.org/
7. Aston, P.: The Grinder, a Java Load Testing Framework. http://grinder.sourceforge.net/
8. Chen, S., Moreland, D., Nepal, S., Zic, J.: Yet Another Performance Testing Framework. In: Australian Conference on Software Engineering (ASWEC). (2008) 170 –179
9. Zhang, L., Chen, Y., Tang, F., Ao, X.: Design and Implementation of Cloud-based Performance Testing System for Web Services. In: Conference on Communications and Networking in China (CHINACOM). (2011) 875 –880
10. Amazon Web Services LLC: Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/ (2012)
11. Cramer, R., Damgård, I., Nielsen, J.: Multiparty Computation, an Introduction (2009)
12. Freedman, M., Nissim, K., Pinkas, B.: Efficient Private Matching and Set Intersection. In: Advances in Cryptology (EUROCRYPT). Volume 3027 of LNCS., Springer (2004) 1–19
13. De Cristofaro, E., Tsudik, G.: Practical Private Set Intersection Protocols with Linear Complexity. In: Financial Cryptography and Data Security (FC). Volume 6052 of LNCS., Springer (2010) 143–159
14. Micciancio, D.: A First Glimpse of Cryptography's Holy Grail. Commun. ACM **53**(3) (March 2010) 96–96
15. Paillier, P.: Public-key Cryptosystems Based on Composite Degree Residuosity Classes. In Stern, J., ed.: Advances in Cryptology (EUROCRYPT). Volume 1592 of LNCS., Springer (1999) 223–238
16. Damgard, I., Jurik, M.: A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In: Public Key Cryptography. Volume 1992 of LNCS. Springer Berlin / Heidelberg (2001) 119–136
17. IEEE: Standard for System and Software Verification and Validation (2012)
18. Reenskaug, T.: Model-Viewer-Controller. Technical report, XEROX PARC (1978)
19. The Pylons Project: Pyramid. http://www.pylonsproject.org/ (2012)
20. SQLAlchemy Authors and Contributors: SQLAlchemy. http://www.sqlalchemy.org/
21. Mako Authors and Contributors: Mako Templates for Python. http://www.makotemplates.org/
22. MySQL AB: MySQL - The World's Most Popular Open Source Database. http://www.mysql.com/ (2011)
23. Crockford, D.: The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational) (July 2006)
24. Hunter, J.: Matplotlib. http://matplotlib.sourceforge.net/ (2011)