Architecture, Implementation and Application of Tools for Experimental Analysis

Tom Dowling, and Adam Duffy

***Abstract*—** This paper presents an architecture to assist in the development of tools to perform experimental analysis. Existing implementations of tools based on this architecture are also described in this paper. These tools are applied to the real world problem of fault attack emulation and detection in cryptographic algorithms.

***Keywords*—** Software Architectures and Design, Software Components and Reuse, Engineering Secure Software.

I. Introduction

TYPICALLY, experimental analysis is performed using software with an ad hoc design. This approach reduces the ability of the tool and its components to be modified for use in other types of experiments. It is also difficult to combine these tools so as to enable more sophisticated experiments and analysis to be performed.

To overcome these problems this paper proposes an architecture for the development of experimental analysis tools. This architecture, as described in section II, is a generalisation of an existing architecture [22]. The architecture is also influenced by the xUnit architecture [15].

Through applying the proposed architecture developers will be able to create tools for experimental analysis that can evolve and be combined with other tools with the same architecture. Section III contains details of an experimental analysis tool for obtaining execution times. A benchmarking application, that has been refactored to adhere to the architecture, is presented in section IV. Section V gives an overview of a tool for emulating fault attacks.

Section VI demonstrates how the three experimental analysis tools just mentioned were applied to evaluate both the success and cost of error detection mechanisms within code. Error detection mechanisms are checks embedded into code that are used to detect faults such as errors in data.

II. An Architecture for Experimental Analysis Tools

The Experimental Analysis Tool Architecture is shown in figure 1. It describes a tool that consists of eight subsystems. The Controller coordinates all the activities of the other subsystems. The Experiment Manager stores the experiments to be performed. The Activator starts the system under test. The Experimenter performs the experiment on the system and the results of the experiments are recorded by the Reporter. The Analyst evaluates the results of the experiment and the Presenter creates graphs and tables based on the analysis. The User Interface allows the user to select the types of experiments, the analysis to be performed and the manner of reporting and presenting the results.

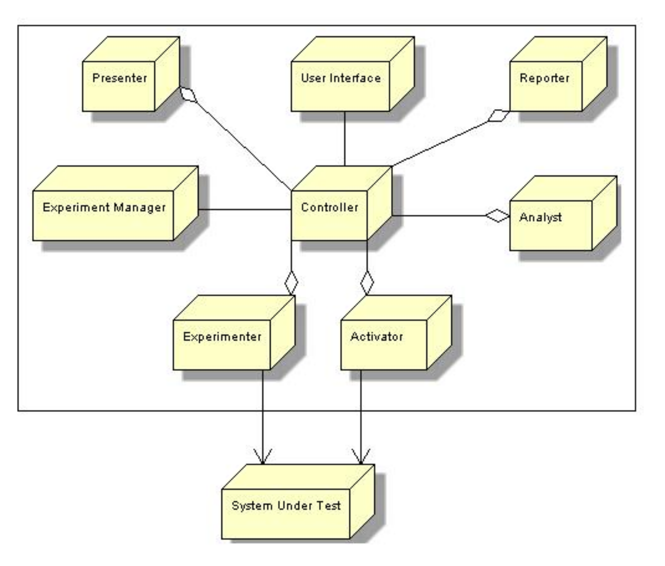


Fig. 1 The Experimental Analysis Tool Architecture

A possible sequence of interactions is as follows. The Controller obtains from the User Interface the information concerning all the experiments to be performed. This information is then passed from the Controller to the Experiment Manager. The Controller then requests from the Experiment Manager the required information for one experiment and instructs the Activator to start the system, or subsystem, that the experiment is to be performed on. Following this, the Controller directs the Experimenter to perform the relevant experiment. Upon completion of the experiment, the Experimenter reports the relevant data to the Controller that records this information using the Reporter. The Controller then requests the next experiment from the Experiment Manager and repeats the sequence above.

When all the experiments have been carried out, the Controller triggers the Analyst to perform the desired analysis of the data obtained during the experiments. Note that the architecture is flexible enough to allow for real time analysis of the experimental results - in this case the Controller would trigger the Analyst before commencing the experiments. The results obtained by the Analyst are then returned to the Controller to be recorded by the Reporter and passed to the

Authors are with Computer Security & Cryptography Group, Computer Science Department, National University of Ireland, Maynooth, Co. Kildare, Ireland (e-mail: cryptogrp@cs.nuim.ie).

Presenter. The Presenter produces the necessary graphs, tables and other representations required by the end user of the tool to assist in his/her evaluation of the system under test. The Reporter and Presenter subsystems may send information to the User Interface, via the Controller, in the case of a Graphical User Interface. The Reporter and Presenter subsystems are similar in that they are responsible for handling the output of the experiments performed. However, they differ in that the Reporter will record all the data resulting from the experiment, including results that may not be easily comprehended by the end user, and the Presenter will present the data, or a subset of the data, and its analysis in human readable format such as graphs.

The architecture proposed here has the benefit of allowing the individual subsystems of the experimental analysis tool, such as the experimenters, analysts, presenters, etc., to be modified without affecting the other subsystems. It also permits the developer of the tool to reuse those subsystems in other experimental analysis tools.

The architecture presented here can be found in three existing tools: a tool for obtaining timings of algorithm implementations, a modified version of the JavaGrande Forum Sequential Benchmark and a fault attack emulation tool. Each of these tools will be described in turn in the following sections.

III. Timing Tool

A unit of code is typically a method or a function. An API for obtaining timings of code units was developed based on the architecture design described in this paper. Further information about this tool can found in [9].

Figure 2 illustrates how the Timing Tool follows the Experimental Analysis Tool Architecture.

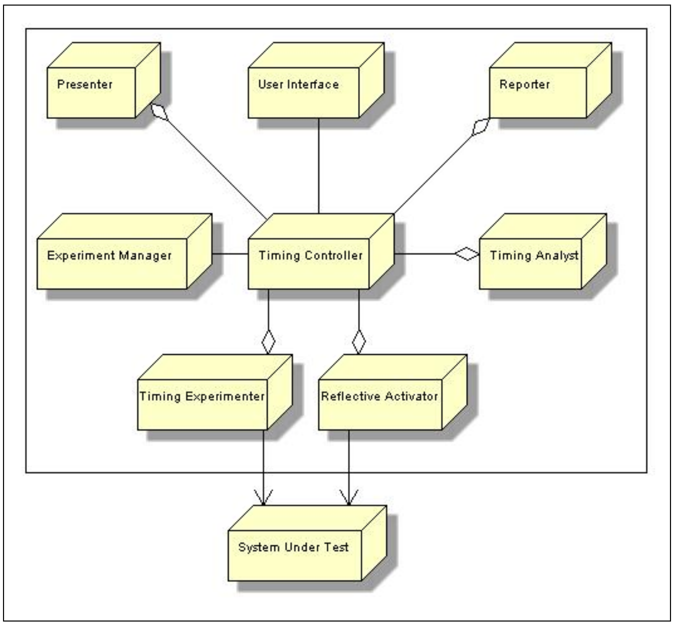


Fig. 2 The Timing Tool following the Experimental Analysis Analysis Tool Architecture

The Reflective Activator was created to invoke methods using the Java Reflection API [11]. The command line based user interface accepts a flat file containing the experiments - the instances and methods of those instances that timings are required of. The Experiment Manager passes all the relevant information for each experiment to the Reflective Activator, by way of the Controller, that then creates and invokes those instances.

The Experimenter consists of a timing mechanism, the Stopwatch. The Stopwatch mimics the functionality of a conventional physical stopwatch. Note that the Stopwatch cannot be more precise than the underlying operating system. A threshold parameter is used to ensure that all timings exceed the granularity of the operating system by a significant amount. The experiment is performed repeatedly until the threshold is reached.

The total running time and number of times the system under test was invoked are reported, not just the average running time. This would allow for the calculation of the standard deviation, variance and other statistical calculations. Consequently, statistical decision theory could be employed to determine if improved timings of enhanced code are general or just specific to the test data.

A disadvantage of this approach to obtaining timings is that the result may include more than the code's execution time. Time used by other processes on the system or incurred during communication within the tool may result in inaccurately high timings. To resolve this an error value is calculated by invoking an empty method the same number of times as the system under test was invoked during the experiment and observing the time taken.

The results of the experiments are recored using the Reporter API. This API is based on the Observer Design Pattern [12, §5.7]. An instance of the Dispatch Reporter contains a list of all the reporters used by the Reporter subsystem. When a result is reported, the Dispatch Reporter passes the result to all the other reporters, as determined by the user of the tool, so that the results can be published in eXtensible Markup Language (XML), Comma-Separated Value (CSV), a Graphical User Interface (GUI) and other formats at the same time. The Permission Reporter allows the user to select the information to be reported to a particular report such as, for example, the GUI Reporter where not all the information may be required by the user.

A simple Analyst was created to obtain the average running times. Presenters were implemented to create graphs using Matlab [17] and tables suitable for TEX [13], [21] (as shown in table I). Automatically generating the graphs and tables ensures the consistency between the different forms of result presentations.

IV. Benchmarking Tool

The JavaGrande Forum[1] (JGF) Sequential Benchmark[2] is a benchmarking tool for the Java Virtual Machine (JVM). This benchmark was selected due to the availability of the

[1] http://www.javagrande.org/
[2] http://www.epcc.ed.ac.uk/javagrande/

TABLE I

THIS TABLE WAS CREATED FROM AN AUTOMATICALLY GENERATED TEX FILE BASED ON THE RESULTS OBTAINED BY TIMING A COMPONENT OF AN APPLICATION. NOTE THAT THE RESULTS DISPLAYED HERE CORRESPOND TO THOSE IN FIGURE 4(A)

	512 version 1	384 version 1	256 version 1	224 version 1	192 version 1	160 version 2	160 version 1	128 version 2	128 version 1	112 version 2	112 version 1	average	rank
Affine	3.14	2.08	1.17	1.02	0.86	0.68	0.88	0.56	0.56	0.45	0.46	0.0047	1
Projective	3.62	2.35	1.37	1.19	1	0.79	0.8	0.65	0.65	0.5	0.49	0.0053	2
Jacobian	4.23	2.83	1.65	1.45	1.21	0.94	0.94	0.77	0.77	0.62	0.61	0.0063	4
Chudnovsky	5.26	3.46	2.07	1.78	1.46	1.17	1.16	0.95	0.93	0.76	0.75	0.0078	6
Jacobian and Affine	4.12	2.64	1.53	1.32	1.13	0.87	0.87	0.72	0.71	0.56	0.59	0.0059	3
Jacobian and Chudnovsky	4.67	2.99	1.76	1.51	1.28	1.01	1	0.83	0.82	0.66	0.67	0.0068	5

source code that allows users to understand precisely what the benchmark is testing. Furthermore, the JGF benchmarks are available free of charge [4].

The JGF Sequential Benchmark consists of three sections. Each section consists of many experiments. Timings are obtained for each experiment. Analysis is performed on these timings to obtain a value, referred to as the JGF number, relative to a reference system. Further information about the JGF benchmarks may be found in [5].

The benchmark has been refactored by the authors to conform to the architecture pattern just described. Beforehand, a user of the benchmark had to run each section of the benchmark separately and in its entirety. Alternatively, the user had to write customized code to perform a subset of the benchmark. It is now possible to perform a custom made selection of benchmark tests using configuration files. Functionality was added to allow the benchmark to be interrupted and resumed without the need to rerun experiments that had already been completed.

The results of the benchmark were reported to the console only. It was necessary to manually redirect this output to a file before any analysis could be performed. This could result in the loss of data. The results were also interpreted before being reported to the console - an approach not in keeping with scientific standards of observation and measurement. The benchmark now uses the Reporter API to record the results in the desired output formats.

The analysis of the JGF benchmark resulted in the generation of a HTML file. This file simply contained a series of tables containing the JGF numbers, obtained during the analysis, in a plain and unintuitive format. By reusing the Presenters created for the timing tool it is possible to compare, at a glance, the performance of two or more different machines to the benchmark machine (see figure 3).

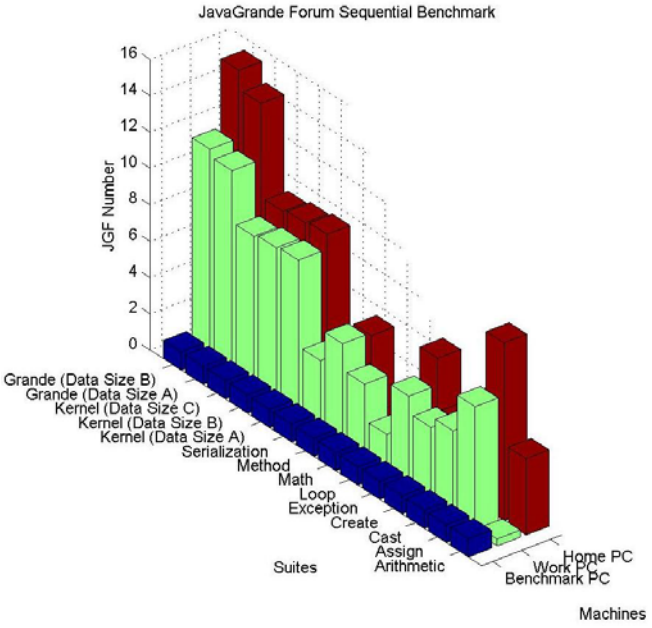


Fig. 3 A graph showing the results of the JGF Sequential Benchmark for two machines relative to the benchmark machine

V. FAULT ATTACK EMULATION TOOL

Researchers tend to develop software fault injection tools to emulate fault attacks rather than obtain expensive specialized hardware [16]. The authors have developed a fault attack emulation tool based on the experimental analysis architecture pattern using the Java Debugger Interface (JDI). The JDI is one component of the Java Platform Debugger Architecture[3] (JPDA) included in the Java 2 Standard Edition (J2SE) platform. The JPDA is intended for developers to create end-user debugger applications but has been appropriated for our purposes. This is a common approach in developing software fault injection tools [20].

The advantage of using the JDI as a basis for a software fault injection tool is that it removes the need for any modification of the code under test. It also allows the entire software fault injection tool to operate on a different machine, virtual or physical. A disadvantage of using JDI is the lack of documentation on the topic.

The Reflective Activator, developed for the timing tool, was extended to create a Standalone Reflective Activator. This enabled the activator to run the system under test on a separate virtual or physical machine - a necessary requirement for using the Java Debug Interface (JDI). The faults are injected into variables within the system during its execution.

An Experimenter, the Reflective Fault Injector, was created for this tool. It performs the actual injection of the faults into the system. It uses the Reflection API to determine the type of a specific variable and to inject a fault into a primitive variable within that variable's object. For example, the `BigInteger` class used in many cryptographic applications uses an array of integers to store its value. The fault is injected by flipping a single bit of an integer within the array. The recursive nature of the Reflective Fault Injector allows for high level classes, such as those representing points on an elliptic curve, to be injected with faults without the need to create any specialized code to handle those high level classes.

[3] `http://java.sun.com/products/jpda/`

The remainder of the subsystems of the tool are reused from the previously mentioned tools. Figure 5(a) shows the number of faults injected into a system under test that had an effect on the result of the system. It also shows that none of the faults were detected. This was due to the lack of Error Detection Mechanisms (EDMs) within the code of the system under test.

VI. A Practical Application of the Experimental Analysis Tools

This section contains a practical application of the architecture to a real world problem. The tools are combined to emulate fault attacks and to evaluate the effectiveness and cost of mechanisms designed to detect those attacks.

Fault analysis uses calculation faults that are deliberately generated or occur naturally within a program. These attacks may be targeted attacks to obtain secret information [3] or nuisance attacks to corrupt the results of the software [14], [25]. For a broad survey of cryptographic fault attacks and suggested countermeasures see [1].

In order to detect faults within an application the developer must introduce Error Detection Mechanisms (EDMs) into the software. A generalized approach to including EDMs in source code was proposed in [23]. It is based on the Simple Duplication with Comparison (SDC) countermeasure where every variable is duplicated and compared after every read operation. The comparison after every read operation minimises the error latency time - the time between when the fault occurs and when it may have an effect on the system under test.

While the SDC EDM is a very general countermeasure there exists alternative EDMs that are appropriate for specific applications. An example of this is the Elliptic Curve Point Validation (ECPV) EDM for an Elliptic Curve Cryptosystems (ECC). Further details of elliptic curves and their use in cryptography can be found in [2], [18].

The authors of [7] describe a possible attack on an ECC through the injection of faults into the elliptic curve arithmetic operations of the cryptosystem. The attack allows for the recovery of the private key by a malicious third party. In the same paper a technique, the ECPV EDM, to detect this attack is presented. The points $P(x, y)$ used in the elliptic curve arithmetic are verified using the curve parameter a_6 such that $y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x = a_6$. If the point passes this check then it is a valid point on the curve. If it fails then a fault has been detected and an exception is thrown.

Of the one hundred and ten attempts to inject a fault into an operation only a certain number will have an affect on the result of the operation. This may be due to the fault being injected into a variable at a point in time where it is no longer in use by the implementation. The faults injected into the implementation that do affect the result of the operation are considered to be effective faults. The success of an EDM is determined by the number of the effective faults that are detected. Of interest to the authors is the question of whether the ECPV EDM or SDC EDM is better with regards to speed and to the rate of success at detecting faults.

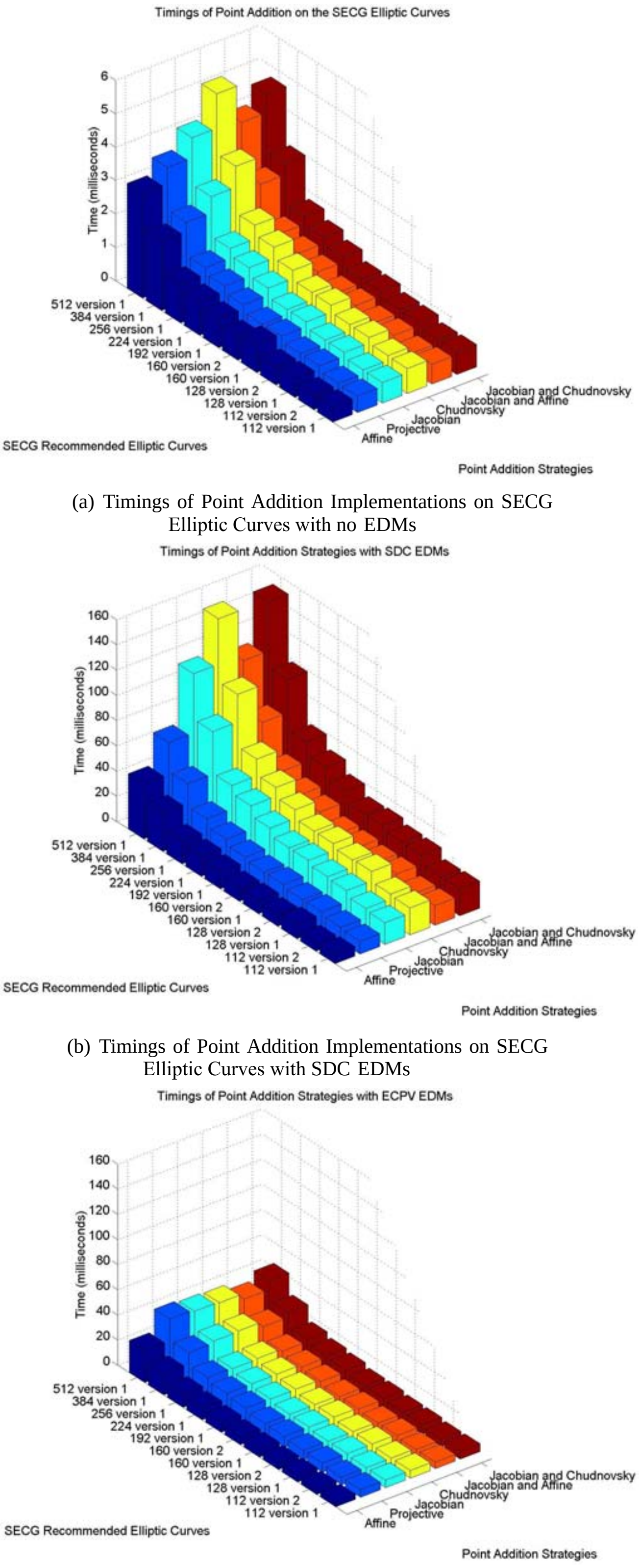
The cost of using the SDC EDM approach in point addition implementations, where a point addition is one of the basic arithmetic operations in ECC, is shown in figure 4(b). A cursory comparison of figures 4(a) and 4(b) shows a significant increase in the running times of the arithmetic operation taking from 10 to 15 times longer to run. Figure 4(c) shows the cost of using the ECPV EDM is approximately half that of SDC. In terms of speed, it is preferable to use ECPV EDMs to detect faults in operations of ECC implementations rather than SDC EDMs. The question remains, which EDM has a better success rate at detecting faults?

However, it is not immediately clear if ECPV EDM (figure 5(c)) is better, worse or the same as SDC EDM (figure 5(b)) in catching faults that occur within implementations. Also, the results are just for a sample of test data based on the SECG recommended elliptic curve parameters. Given that implementations with ECPV EDM are faster than those with SDC EDM, it is desirable to know if ECPV EDM is as capable as detecting faults as SDC EDM. To this end, statistical decision theory was used to determine the answer.

A Difference of Proportions Test of Significance Analyst component was developed to be used by the Fault Attack Emulation tool. This component accepts as input the results of emulating fault attacks on implementations with SDC EDMs and ECPV EDMs. It carried out the calculations required for the test of significance and determined that ECPV EDMs have a greater success rate in detecting faults than the SDC EDMs.

Therefore, the ECPV EDMs are both faster and have a better success rate than SDC EDMs. However, the ECPV EDMs are specific to implementations of elliptic curve arithmetic whereas the SDC EDMs can be used in any implementation. Furthermore, the SDC EDMs may be extended to Multiple Duplication and Comparison (MDC) EDMs. In MDC EDMs every operation is performed thrice and a majority vote taken in the event of a fault allowing the EDM to correct the fault and continue performing the operation.

Further details of the above experiment and the statistical analysis of the results may be found in [10]. Please consult [8], [24] for more information on statistical decision theory.

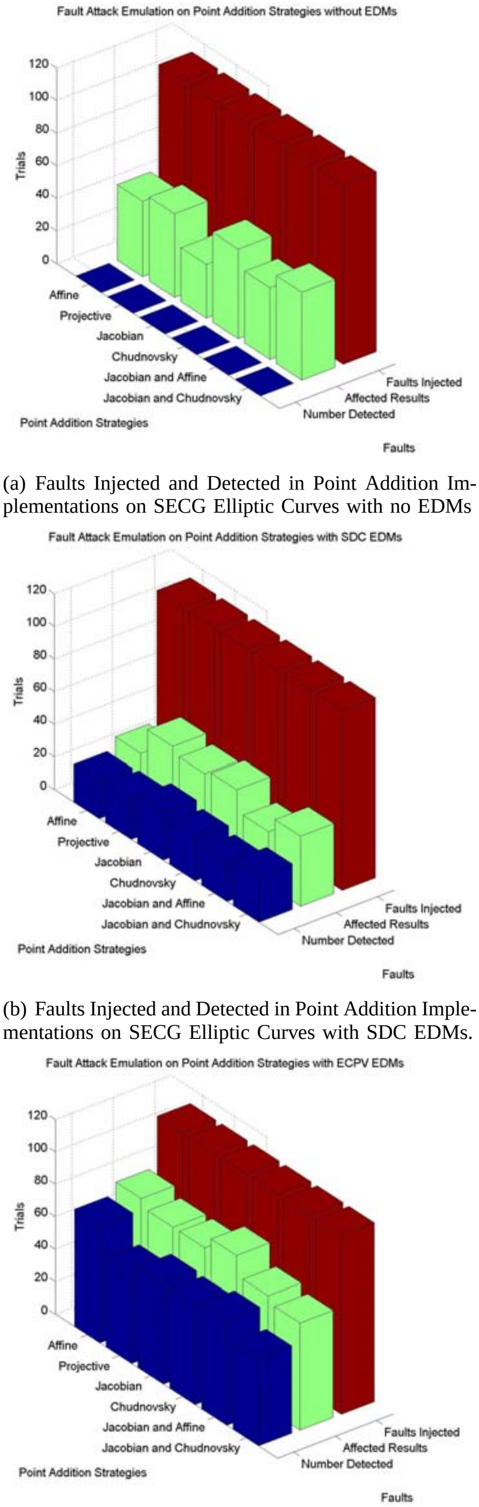


(a) Timings of Point Addition Implementations on SECG Elliptic Curves with no EDMs

(b) Timings of Point Addition Implementations on SECG Elliptic Curves with SDC EDMs

(c) Timings of Point Addition Implementations on SECG Elliptic Curves with ECPV EDMs

Fig. 4 Timings of Point Addition Operations on the SECG Elliptic Curves



(a) Faults Injected and Detected in Point Addition Implementations on SECG Elliptic Curves with no EDMs

(b) Faults Injected and Detected in Point Addition Implementations on SECG Elliptic Curves with SDC EDMs.

(c) Faults Injected and Detected in Addition Implementations on SECG Elliptic Curves with ECPV EDMs

Fig. 5 Faults Injected and Detected in Point Addition Operations on the SECG Elliptic Curves

VII. Conclusions and Future Work

This paper presented an architecture pattern for the development of tools for experimental analysis of algorithm implementations. It also described three tools built according to this architecture pattern and demonstrated their use in solving a practical problem.

Power analysis attacks measure the power consumed by a specific task to get the secret key [19]. By reusing components of the existing tools it our intention to develop a tool to emulate Simple and Differential Power Analysis. This will be achieved through extensive reuse of components of the tools described in this paper. This will enable developers to investigate the susceptibility of their code to these types of attacks.

The modular nature of the framework will assist the integration of the experimental analysis tools into other products. The authors plan to extend the existing tool to enable the timing of operations on the JavaCard platform [6]. This only requires the development of a new Activator, the APDU Activator (APDU stands for Application Protocol Data Unit, the basic unit of communication with a smart card). The rest of the code can remain unchanged. The new tool may also form the basis of an open source JavaCard benchmarking tool.

References

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. *Workshop on Fault Detection and Tolerance in Cryptography*, 2004.

[2] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1st edition, 1999.

[3] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. *Lecture Notes in Computer Science*, 1233:37–51, 1997.

[4] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A Benchmark Suite for High Performance Java. *Concurrency: Practice and Experience*, pages 375–388, 2000.

[5] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. Benchmarking Java Grande Applications. *Proceedings of The Second International Conference on The Practical Applications of Java*, pages 63–73, 2000.

[6] Z. Chen. *Java Card Technology for Smart Cards*. Addison-Wesley, 1st edition, 2000.

[7] M. Ciet and M. Joye. Elliptic Curve Cryptosystems in the Presence of Permanent and Transient Faults. *Designs, Codes and Cryptography*, 36:33–43, 2005.

[8] J. Devore and R. Peck. *Statistics: The Exploration and Analysis of Data*. Duxbury Press, 2nd edition, 1993.

[9] T. Dowling and A. Duffy. A Java API for Experimental Analysis of Algorithms. *Proceedings of IASTED International Conference of Software Engineering*, 2004.

[10] A. Duffy. Experimental Analysis of EDMs in Implementations of Elliptic Curve Cryptography. Technical report, National University of Ireland, Maynooth, 2006. To Appear.

[11] I. R. Forman and N. Forman. *Java Reflection in Action*. Manning Publications Co., 1st edition, 2004.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object Oriented Software*. Addison Wesley, 1st edition, 1994.

[13] M. Goossens, F. Mittelbach, and A. Samarin. *The LaTeX Companion*. Addison-Wesley Professional, 1st edition, 1993.

[14] S. Govindavajhala and A. W. Appel. Using Memory Errors to Attack a Virtual Machine. In *IEEE Symposium on Security and Privacy*, pages 154–165. Institute of Electrical and Electronics Engineers, Inc., 2003.

[15] P. Hamill. *Unit Test Frameworks*. O'Reilly, 1st edition, 2004.

[16] M. C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, 1997.

[17] B. R. Hunt, R. L. Lipsman, and J. M. Rosenberg. *A Guide to Matlab: For Beginners and Experienced Users*. Cambridge University Press, 1st edition, 2001.

[18] N. Koblitz. *A Course in Number Theory and Cryptography*. Springer, 2nd edition, 1994.

[19] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. *Advances in Cryptology - Crypto '99*, pages 388–397, 1999.

[20] N. Krishnamurthy, V. Jhaveri, and J. A. Abraham. A Design Methodology for Software Fault Injection in Embedded Systems. *Proceedings of the IFIP International Workshop on Dependable Computing and its Applications*, pages 237–248, 1998.

[21] L. Lamport. *LaTeX: A Document Preparation System*. Addison-Wesley Professional, 2nd edition, 1994.

[22] N. G. M. Leme, E. Martins, and C. M. F. Rubira. A Software Fault Injection Pattern System. *Proceedings of the 8th Conference on Pattern Languages of Programs*, 2001.

[23] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violente. Soft-error Detection through Software Fault Tolerance Techniques. *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 210–218, 1999.

[24] M. Spiegel. *Theory and Problems of Statistics*. McGraw Hill, 2nd edition, 1972.

[25] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An Experimental Study of Security Vulnerabilities Caused by Errors. In *IEEE International Conference on Dependable Systems and Networks*, pages 421–432. Institute of Electrical and Electronics Engineers, Inc., 2001.