Side Channel Leakage Profiling in Software

Daniel Shumow*and Peter L. Montgomery†

February 1, 2010

Testing cryptographic implementations for side channel leakage is a difficult and important problem. The techniques used to uncover side channel leakage are more involved than the usual methodologies of software testing, for example sometimes involving physical measurements of hardware. As such, it is difficult to work this sort of analysis into the usual software testing process. To this end we have developed the side channel profiler. This is an extensible framework for capturing dynamic execution of cryptographic code and applying side channel analysis regardless of underlying architecture. This tool can be used to selectively emulate different hardware components, or apply other side channel leakage criteria. We also demonstrate how the tool can be used to analyze an implementation of naïve square and multiply modular exponentiation.

1 Motivation

There is not unanimous advice on how to comprehensively prevent or test for side channel leakage from a cryptographic implementation. In [2] and [9], Ferguson et al. and Molnar et al. advocate code that has no conditional branches or memory accesses. However, while this criterion appears quite restrictive, it is not comprehensive. For example, individual instructions may exhibit variable timing or power use ([5] and [6].) There are also theoretical models for analyzing and proving algorithms secure against side channel leakage ([8] and [10].) However, even if a programmer implements a cryptosystem provably secure against side channel leakage, there is a gap between implementation and theoretical models. Theoretical proofs still leave room for implementation specific leakage. Emulation has also been used to evaluate side channel leakage on smart cards ([3] and [4].) However, this approach will work only for simple smart card processors. In general, programmers implementing cryptographic systems are left without comprehensive guidance on techniques to prevent side channel leakage, criteria for side channel security and testing frameworks to validate implementations.

In addition to a lack of comprehensive high level tools and techniques for generally avoiding side channel attacks, reproducing side channel leakage is difficult. A technique that software engineers use in debugging is the "repro" or reproduction of the software bug. However, when the bug is information leakage from a side channel, reproducing the bug in the purest sense consists of actually mounting a side channel attack. This may be prohibitively difficult for programmers developing cryptographic software. As such, these software engineers are deprived of a key technique of debugging their implementations: reproducing the bug.

*Microsoft Research, Redmond, USA. Email: danshu@microsoft.com
†Microsoft Research, Redmond, USA. Email: Peter.Montgomery@microsoft.com

2 Side Channel Profiler: A Software Solution

We have developed the side channel profiler, a flexible extensible framework for testing cryptographic implementations against side channel attacks. This provides a methodology for evaluating side channel information leakage of cryptographic implementations that is consistent with other software testing techniques. Furthermore, this tool can be used by engineers who are not experts in side channel attacks, yet remains useful for experts. The side channel profiler is a framework including both a development API as well as a program for evaluating side channel attacks.

The profiler works similarly to an emulator for more advanced processors than smart cards. However, modern computer processors, even for embedded systems, are too complex to emulate in the detail needed with any sort of reasonable performance. This is not to say that in general emulation is too slow. Indeed, recent trends in virtualization, such as those applied to cloud computing, show that emulation technology can obtain high performance. The problem for evaluating a piece of software for side channel leakage is that the goal is not to merely emulate another processor. Rather, the purpose is to execute a piece of cryptographic software and simultaneously analyze its behavior. As such, for every instruction executed by the program being evaluated, the side channel profiler must halt execution and perform an analysis on the state of the processor. To solve this problem, the side channel profiler does not simulate the entire state of the processor; rather it selectively simulates or analyzes specific leakage according to an analysis module.

The side channel profiler operates as follows: The user selects a cryptographic implementation, a program that will exercise this implementation and an analysis module. The cryptographic implementation is specified by the binary (the crypto module) and the name of the specific function to evaluate. The program that exercises this implementation can be any program that causes the specified function to be exercised; this can be a simple test program or an actual security component that uses the cryptographic implementation. The profiler runs the program and traces the processor state at the execution of each instruction. This information is fed into an analysis module that, upon completion of execution, generates a report on side channel leakage.

2.1 Architecture and Implementation

The side channel profiler contains three main architectural components: the core framework, the side channel profiling program and the analysis modules. The core framework consists of an engine and an API by which this engine is controlled. The engine controls the execution of the cryptographic implementation and passes this information to the analysis plugins. The API is used to specify the target cryptographic implementation, as well as the analysis modules to use as well as other optional information, such as output files. The side channel profiling program is a simple user interface to the side channel profiling framework. The analysis modules consist of a set of included plugins, presently only modules that evaluate control flow or memory accesses. However, these modules can be user defined and plugged in. The various architectural components and data flow between them are succinctly described in figure 1.

The primary architectural feature of the side channel profiler is that it provides a plug in point for new analysis modules. Before execution of each instruction, the analysis module is passed the state of the processor and records any relevant information. This allows experts to write new analysis modules to evaluate implementations against new attacks. Furthermore, this allows users of the framework to select the type of leakage

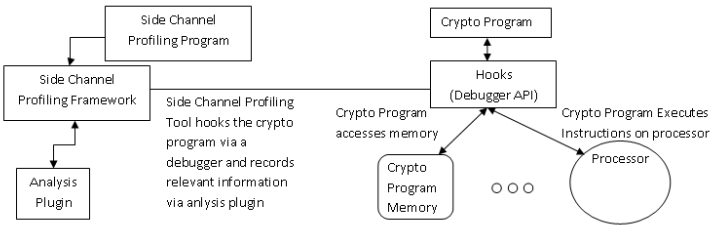


Figure 1: Architectural Components and Data Flow.

to evaluate their implementation for. This pluggable architecture allows the tool to flexibly accommodate new types of attacks.

In our current implementation, we use the Microsoft Debugger API to trace the processor state, as well as the state of the process memory. However in principle, any tracing utility that support instruction- and register-level tracing, such as other debuggers, would work as well. The Microsoft Debugger API consists of the debugger engine, that loads and controls execution of a debuggee process, as well as a set of functions that allow a program to control a debuggee process via this engine. The side channel profiler framework core allows debuggee execution to continue for the execution of one instruction, then transitions control to an analysis plugin. The analysis plugin obtains data about the current processor state via the debugger API, and upon completion returns control back to the core of the framework. This process repeats until the cryptographic implementation completes execution.

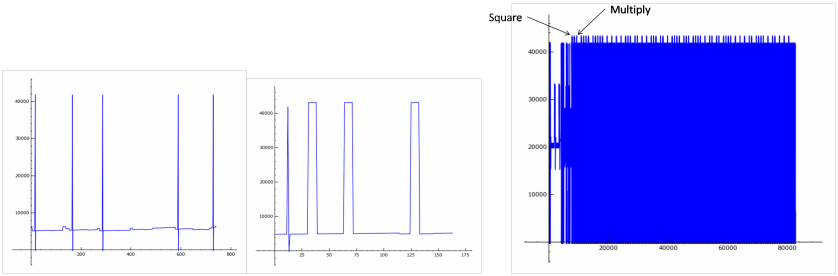
2.2 Analysis Modules

Analysis modules are broadly defined as the pluggable component of the profiler used to analyze the execution of a cryptographic implementation. This is necessarily so, as this component needs to be defined abstractly enough so that if a new attack is discovered, a new analysis module can be easily implemented to test for it. To accommodate the flexible architectural nature of the analysis module, the interface is simple and very flexible. There are three interface points to an analysis module: an initialization function, an analysis function, and an uninitialize function. The initialization function takes a single parameter, an output file for analysis data. The analysis function is passed a reference to the debugger engine and obtains data on the state of the processor by calling debugger API functions. The analysis function is invoked after each instruction execution. Upon return from this function the side channel profiler core executes the next instruction and then invokes the analysis function again. When the side channel profiler completes execution it calls the uninitialize function. This function completes any analysis and writes data to the output file.

There are two broadly defined categories of analysis modules, hardware simulation modules and idealized model analysis modules.

2.2.1 Hardware Simulation

Analysis modules can perform analysis by simulating the behavior of actual hardware. Modern processors are too complex to efficiently simulate in their entirety. However, to avoid this problem the analysis module can be used to simulate one specific component of hardware. For example, by observing the record of memory accesses, we can



(a) Montgomery Multiplication (b) Squaring (c) Full Exponentiation

Figure 2: Program counter offset traces.

simulate cache hits and misses as to evaluate leakage from this side channel. Similarly, by recording the stream of instructions and branches, we can simulate the state of the branch predictor and evaluate any side channel leakage from it. The side channel leakage from these two specific examples is caused by nonconstant program flow and memory accesses. However, with more sophisticated models of hardware we could simulate other side channels like power usage.

2.2.2 Idealized Leakage Models

In addition to emulating hardware, an analysis model can skip this step and directly analyze the program execution according to an idealized model. This is useful if the underlying hardware is overly complicated or unknown. For example, the execution of an algorithm can be analyzed for compliance with the program counter model [9]. This model asserts that the program counter should advance in the same manner while executing a cryptographic algorithm, regardless of input. A similar model holds that memory access should not be conditional on input. To evaluate an implementation against these models, we can directly record the execution of the program across inputs. These records can be compared and used to determine if program flow or memory access is dependent upon secret data. Aside from these two examples, the architecture is flexible enough to allow us to implement analysis modules for other idealized leakage models as well.

3 Case Study: Naïve Square and Multiply

We ran the side channel profiler on a Naïve square and multiply modular exponentiation routine. This is not a surprising example of side channel vulnerability, but it is demonstrative of the capabilities of the profiler. To this end we implemented Naïve square and multiply exponentiation, compiled with visual studio 2008 for x86 (32 bit) architecture. We implemented a simple program to run a square and multiply modular exponentiation routine. The analysis module that we used for this test case evaluates the execution with the program counter model [9] and records the program counter as it runs through the exponentiation computation.

Figure 2 shows the output of this side channel profiler after analyzing the execution of exponentiation modulo a 64 bit prime with a 16 bit exponent. The profiler can handle inputs of a more cryptographically interesting size, but the visualization is clearer at

these lengths. Figures 2a and 2b shows the traces of the offset of the program counter from the base address of the module (dll or executable.) The places where the trace spikes down to -1 indicate places where program flow transferred out of the module containing the crypto code. These portions of the record have been cut out solely for visualization purposes. This is not problematic for analysis though, these calls outside are to the Microsoft visual studio C run time library for memset and have constant program flow. The program counter trace for the whole exponentiation is shown in figure 2c; the pattern of squares and multiplies is visible in the trace. The tool does not expose any variation in program flow in the squaring or multiplying routines. By obtaining the trace record, and performing pattern matching on the multiply and squaring routines, we can automatically recover the exponent that was used. Although this example is very simple, it is a very clear demonstration of the concept and most basic capabilities of the side channel profiler.

4 Future Work

The previous section shows that side channel profiler is able to analyze cryptographic implementations. However, it is a work in progress with much room for improvement and future development. There is room for both improvement of the architecture and implementation as well as a significant amount of work on building analysis modules.

4.1 Architectural and Implementation Improvements

The main architectural and implementation improvement for the side channel profiler is a performance improvement. Although sufficient to perform analysis, the analysis for large cryptographic sizes (in the thousands of bits) can take hours. To maximize the efficiency of the test and development cycle this performance needs to be improved.

Presently, as the side channel profiler is implemented, the framework passes the state to the analysis plugin at each instruction execution. It may be the case that the analysis module does not actually need information from each instruction. In the current design, if this is the case, the analysis module just returns control back to the framework. However, the act of stepping through a process instruction by instruction is not optimal. In so far as, transitioning control from the debugger engine to the debuggee actually takes a significant amount of time, and may be unnecessary for all analysis plugins.

To prevent the debugger engine from unnecessarily taking control of the target process the analysis module needs a way to specify when it should be invoked by the side channel profiler next. Specifically, the analysis module should optionally specify a filter that the side channel profiler uses to determine the break points to transition control from the target process. This way the debugger engine only halts the target process when necessary.

4.2 Implement Advanced Hardware Analysis Modules

At this time, the side channel profiler does not include any specific hardware analysis modules. While our framework makes this possible, it remains a difficult problem. Indeed, to make an analysis module that models hardware, it would be necessary to have access to the actual design of the hardware. In fact, modeling the entire processor on this level may be infeasible. However, modeling the entire processor may not be necessary, nor is modeling the entire processor on each instruction. Rather, it may be

sufficient to model only particular processor components, such as the arithmetic logical unit or the multiplier.

In the future, we would like to apply these ideas to write an analysis module to perform power analysis. Specifically, with a model of the multiplier and its power usage profile we can write an analysis module that ignores all other instructions besides multiplies. On multiplies, this analysis module would simulate the operation of the hardware and its power usage profile. Upon completion of the cryptographic operation the record of multiplies and their power usage profile can be analyzed for leakage. Of course, such a profile would not give us general assurance that our implementation does not leak information through power usage channels. However, it could test for leakage through power channels during multiplies. In further refinements of this module, we can target power usage of the hardware components that are most likely to leak information by this channel.

5 Conclusion

The side channel profiler is a software based solution to test cryptographic implementations for side channel leakage. We implemented the tool on the windows platform using the Microsoft debugger API, but similar techniques can work with other tracing utilities. We demonstrated a simple analysis of square and multiply modular exponentiation in the program counter model using the tool. This simple example shows the basis for how this tool can be extended to analyze more complex and realistic implementations of cryptographic algorithms.

This tool, in and of itself, is not a general method to detect all types of side channel leakage. The approach described here has only been shown detecting side channel leakage that is caused by data dependent variation in execution flow and memory accesses. However, the architecture is pluggable and allows for adding new capabilities to test for new types of threats. As such, it can be adapted to detect other types of leakage, thus providing a software tool for a testing process to detect and eliminate more general types of side channel leakage.

References

[1] O. Acıiçmez, Ç. K. Koç and J.P. Seifert. *On the Power of Simple Branch Prediction Analysis*, Cryptology ePrint Archive, Report 2006/351, 2006.

[2] N. Ferguson and B. Schneier. *Practical Cryptography*, Wiley, (2003).

[3] J. den Hartog, J. Verschuren, E. de Vink, J. de Vos and W. Wiersma. *PINPAS a tool for power analysis of smartcards*, Proceedings of SEC 2003, Wolters-Kluwer, pp. 5 (2003).

[4] G. Hollestelle and W. Burgers and J.I. den Hartog. *Power analysis on smartcard algorithms using simulation*, Technical Report Eindhoven University of Technology, http://doc.utwente.nl/66569/, (2004).

[5] P. Kocher. *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*, Advances in cryptology-CRYPTO 1996, Springer Lect. Notes in Comp. Sci. 1109, Springer-Verlag, pp. 104–113 (1996).

[6] P. Kocher, J. Jaffe and B. Jun. *Differential Power Analysis*, Advances in Cryptology-CRYPTO 1999, Springer Lect. Notes in Comp. Sci. 1666, Springer-Verlag, pp. 388–397 (1999).

[7] P. Kocher, J. Jaffe and B. Jun. *Introduction to Differential Power Analysis and Related Attacks*, Technical Report, http://www.cryptography.com/resources/whitepapers/DPATechInfo.pdf, 1998.

[8] S. Micali and L. Reyzin. *Physically Observable Cryptography*, TCC 2004, Springer Lect. Notes in Comp. Sci. 2951, Springer-Verlag, pp. 278–296 (2004).

[9] D. Molnar, M. Piotrowski, D. Schultz and D. Wagner. *The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks*, Cryptology ePrint Archive, Report 2005/368, 2005.

[10] M. Naor and G. Segev. *Public-Key Cryptosystems Resilient to Key Leakage*, Advances in Cryptology-CRYPTO 2009, Springer Lect. Notes in Comp. Sci. 5677, Springer-Verlag, pp. 18–35 (2009).

[11] C. Percival. *Cache Missing for fun and profit*, http://www.daemonology.net/papers/cachemissing.pdf, (2005). Originally presented at BSDCan '05, May 2005.