Extended Abstract: An Experimental Study of a Bucketing Approach*

Yuri Gil Dantas
TU Darmstadt
Darmstadt, Germany

Tobias Hamann
TU Darmstadt
Darmstadt, Germany

Heiko Mantel
TU Darmstadt
Darmstadt, Germany

Johannes Schickel
TU Darmstadt
Darmstadt, Germany

<lastname>@mais.informatik.tu-darmstadt.de

1 Introduction

When a secret has influence on the timing of a program, an attacker can measure the execution time of the program in order to learn some information about the secret. More specifically, this can be done by sending ordinary inputs to the program and analyzing the time taken to execute the program. Traditionally, these attacks, namely Timing Side-Channel Attacks [2], are carried out against cryptographic implementations [2, 13] and web applications [1, 6]. Indeed, there have been several attacks developed against TLS protocol [2], AES [5] and RSA implementations [11], where researchers demonstrated the feasibility of fully recovering the secret key.

Although several approaches [3, 14, 12, 8, 7] have been proposed in order to eliminate timing side-channel attacks, the problem is still not solved, mainly due to practicality and effectiveness reasons. For instance, implementations based on the *static transformation* [8] approach are not fully practical due to the large performance penalty caused by the transformation. Moreover, *dynamic transformation* [7] is not always effective as demonstrated in [4].

Eliminating timing side-channel attacks is challenging, as countermeasures should not only eliminate these attacks by reducing the amount of information leakage from the program, but also should be practical to use. With this in mind, another approach, namely Bucketing [14, 9], has been proposed. Bucketing is a quantitative approach for reducing timing side-channel attacks by decreasing the number of possible timing observations, while minimizing the performance penalty. Although Bucketing has been shown to be sound, it has not been implemented to the best of our knowledge. In this paper, we provide an implementation of Bucketing at the application level. More concretely, we implement Bucketing using a runtime enforcement tool and experimentally evaluate the effectiveness of our implementation for reducing timing side-channel attacks. In summary, the contributions of this paper are two-fold:

- We implement Bucketing at the application level using a runtime enforcement tool. Our implementation is generic in the sense that it can be applied to any Java program with deterministic timing behavior, which is a foundational assumption of Bucketing [14].
- We evaluate the effectiveness of our implementation. For this, we carry out several experiments, with and without using Bucketing. In each experiment, we measure the running time of the program for different secret input values. For all experiments, we observed a quantitative reduction of information leakage from the program when using our implementation.

This paper is organized as follows. Section 2 introduces the concept of Bucketing. Section 3 explains briefly how we implemented Bucketing using a runtime enforcement tool, and Section 4 contains our experimental results. Finally, in Section 5, we conclude the paper by discussing future work.

*This work has been funded by the DFG as part of the project Secure Refinement of Cryptographic Algorithms (E3) within the CRC 1119 CROSSING.

Submitted to:
QAPL 2017

2 Bucketing Approach

Bucketing is a quantitative approach that allows one to discretize the execution time of a program in a way that the results of the computation are only returned at a small number of fixed points in time [14, 9]. That is, Bucketing aims to split all critical output values of a program into buckets such that each output has to wait until the enclosing bucket's upper bound time to be released.

Instead of describing the complete detail of Bucketing, which we refer to [14], we describe its behavior by means of an example (depicted in Figure 1). Assume a program that leaks sensitive information when releasing output events such that an attacker can make (four) different observations about the secret just by measuring the response time of the program. Next, assume that two buckets are defined, b_1, with an upper bound time of t_{b_1}, and b_2, with an upper bound time of t_{b_2}, where the execution time of secret input 1 is allocated into b_1 and secret inputs 2, 3 and 4 into b_2. As a result, whenever this program runs an operation wrt. secret input 1 (similar for inputs 2, 3, and 4), the output event will be held by b_1 until the execution time reaches t_{b_1}. Considering this scenario, the number of timing observations are reduced from four to two, and consequently the power of the attacker to gain information about this secret is also reduced. Moreover, in comparison to static transformation, Bucketing also keeps the performance overhead of such a program minimal. This can be clearly seen on the right side of Figure 1, since not all secret inputs were allocated to the worst-case execution time (i.e. t_{b_2}).

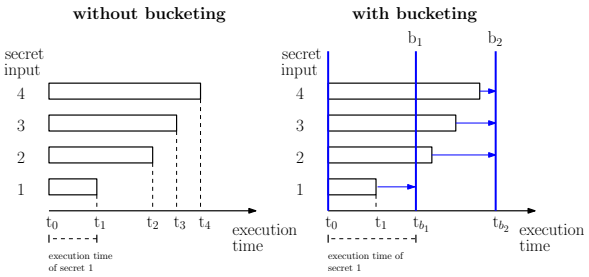


Figure 1: Illustration of a program's timing behavior when releasing sensitive outputs (without Bucketing on the left and with Bucketing on the right).

3 Bucketing Implementation

We implement Bucketing using a runtime enforcement tool, namely CliSeAu [10]. CliSeAu has a modular architecture consisting of four components: interceptor, coordinator, enforcer and local policy. For this particular implementation we just focus on the interceptor and enforcer. *Interceptor* is a component that performs the activity of intercepting attempts of the program to perform security-relevant events and *enforcer* is a component that enforces a countermeasure on the target program. For instantiating CliSeAu for Bucketing, one needs to define the sensitive methods (i.e. code that operates on secret data) of the target program such that CliSeAu can track each call of these methods. Besides, it is also required to instantiate the enforcer by defining the amount of buckets and their respective sizes.

For the sake of space, we only present the sequence diagram (depicted in Figure 2) that describes part of the flow events of our implementation. Firstly, the interceptor intercepts a security-relevant event whenever a sensitive method call is performed by the program. The interceptor sets the initial time of the event and forwards such an event to the enforcer. The enforcer can specify code to execute before and after the security-relevant event. In our implementation, we specify

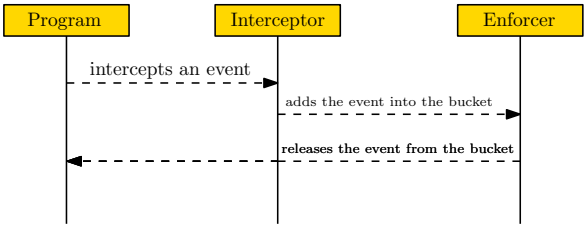


Figure 2: Simplified Diagram of Bucketing implementation

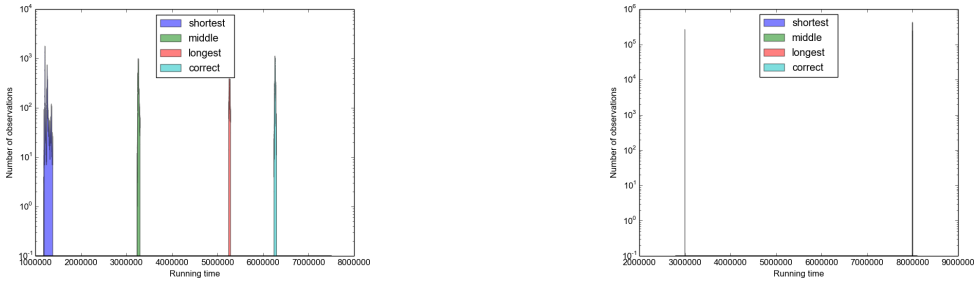
Bucketing for being executed after a security-relevant event such that the enforcer only releases the event when the upper bound time of the current bucket is reached.

Our Bucketing implementation is generic in the sense that it can be (easily) applied to any Java program with deterministic timing behavior, which is a foundational assumption of Bucketing. For more details about CliSeAu's genericity, we refer to [10].

4 Experiments

Our goal is to experimentally evaluate the effectiveness of our Bucketing implementation upon reducing timing side-channel attacks. For the sake of simplicity, we have implemented a simplified example of a client-server application where legitimate clients authenticate (for integrity reasons) their requests into the server using Message Authentication Code (MAC). For this, both legitimate client and server share a common secret key, which is required to build a valid MAC for arbitrary requests. In order to verify a MAC, our server builds its own MAC and compares with the MAC sent by the client. Finally, this comparison is performed by a string comparison method, where we intentionally add delay in four parts of the comparison in order to have a clear timing difference between the responses[1]. As a result, an attacker can explore this timing difference in order to construct a valid MAC. Our assumption is that if the time response of an input_1 takes longer than an input_2 the attacker is closer to guess the correct MAC.

We carry out our experiments as follows. Firstly, we explore the timing side-channel vulnerability of our server by sending four distinct secret inputs, namely *shortest*, *middle*, *longest*, and *correct*. The first three secret inputs (shortest, middle and longest) mean that the first, the middle, and the last character of the MAC are, respectively, incorrect. Moreover, the correct input means that all characters are correct. Figure 3a shows the results of this experiments when not using Bucketing. We can clearly observe differences in the running time values that correspond to four different secret input values. This gives us a hint that an attacker can adaptively infer the expected MAC value by sending arbitrary MACs.



(a) Running time when Bucketing is not applied

(b) Running time when Bucketing is applied

Figure 3: Running time values that correspond to four secret input values

Secondly, we investigate the effectiveness of our implementation upon reducing the timing-side channel vulnerability in our server. For this, we define two buckets of size 3 and 8 ms with the goal of decreasing the power of a potential attacker by reducing her number of timing observations, while taking performance into account, since only one bucket would suffice, in theory, to eliminate the vulnerability. Figure 3b depicts the running time values that correspond to four different secret inputs when Bucketing is applied. On one hand, we can observe that the running time values for the shortest input are always released at around 3 ms (i.e. the upper bound size of the first bucket). On the other hand, we can observe that the other three running time values are overlapping at around 8 ms. Therefore, in contrast to 3a,

[1]This intentional delay simulates programs where the timing differences between the observations are in the range of a few milliseconds rather than nanoseconds. Attacks on programs in this timing range have been shown in e.g. [6].

we cannot observe (much) difference in the running time values that correspond to these three secret input values. We consider these results promising, as they hint on the fact that our implementation of Bucketing is indeed effective in reducing the timing side-channel in our server.

5 Summary and Future Work

This paper provides an implementation of Bucketing at the application level, which is built on our tool for dynamic enforcement of security requirements in Java programs. We carry out a number of experiments for demonstrating its effectiveness. In summary, our experimental results give us a hint that our implementation is effective to reduce timing side-channel attacks. There are many directions for future work, e.g., we are currently investigating how precise and accurate our implementation is, i.e. how close to the actual bucket our implementation releases the information. We are investigating how effective our implementation is to reduce the capacity of timing side-channels [15]. Finally, we are also interested in applying our implementation to a more realistic scenario, where we do not make simplifying assumptions wrt. running time.

References

[1] Martin R. Albrecht & Kenneth G. Paterson: *Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS*. In: *Advances in Cryptology - EUROCRYPT 2016*.

[2] Nadhem J. AlFardan & Kenneth G. Paterson: *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. In: *2013 IEEE Symposium on Security and Privacy, SP 2013*.

[3] Aslan Askarov, Danfeng Zhang & Andrew C. Myers: *Predictive Black-Box Mitigation of Timing Channels*. In: *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010*.

[4] Michael Backes & Boris Köpf: *Formally Bounding the Side-Channel Leakage in Unknown-Message Attacks*. In: *Computer Security - ESORICS 2008*.

[5] Daniel J. Bernstein (2005): *Cache-timing attacks on AES*. Technical Report.

[6] Andrew Bortz & Dan Boneh: *Exposing private information by timing web applications*. In: *Proceedings of the 16th International Conference on World Wide Web, WWW 2007*.

[7] Benjamin A. Braun, Suman Jana & Dan Boneh (2015): *Robust and Efficient Elimination of Cache and Timing Side Channels*. *CoRR* abs/1506.00189.

[8] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere & Bjorn De Sutter: *Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors*. In: *30th IEEE Symposium on Security and Privacy*.

[9] Goran Doychev & Boris Köpf: *Rational Protection against Timing Attacks*. In: *IEEE 28th Computer Security Foundations Symposium, CSF 2015*.

[10] R. Gay, J. Hu & H. Mantel (2014): *CliSeAu: Securing Distributed Java Programs by Cooperative Dynamic Enforcement*. In: *Proceedings of the 10th International Conference on Information Systems Security (ICISS)*, LNCS 8880, Springer, pp. 378–398.

[11] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth & Berk Sunar: *Cache Attacks Enable Bulk Key Recovery on the Cloud*. In: *Cryptographic Hardware and Embedded Systems - CHES 2016*.

[12] Emilia Käsper & Peter Schwabe: *Faster and Timing-Attack Resistant AES-GCM*. In: *Cryptographic Hardware and Embedded Systems - CHES 2009*.

[13] Paul C. Kocher: *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. In: *Advances in Cryptology - CRYPTO '96*.

[14] Boris Köpf & Markus Dürmuth: *A Provably Secure and Efficient Countermeasure against Timing Attacks*. In: *Proceedings of the 22nd IEEE Computer Security Foundations Symposium, CSF 2009*.

[15] Claude E. Shannon: *A mathematical theory of communication*. *Mobile Computing and Communications Review, 2001*.