Anupam Chattopadhyay · Chester Rebeiro
Yuval Yarom (Eds.)

Security, Privacy, and Applied Cryptography Engineering

8th International Conference, SPACE 2018
Kanpur, India, December 15–19, 2018
Proceedings

Springer

Editors
Anupam Chattopadhyay
School of Computer Science and Engineering
Nanyang Technological University
Singapore, Singapore

Chester Rebeiro
Indian Institute of Technology Madras
Chennai, India

Yuval Yarom
University of Adelaide
Adelaide, Australia

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-030-05071-9 ISBN 978-3-030-05072-6 (eBook)
https://doi.org/10.1007/978-3-030-05072-6

Library of Congress Control Number: 2018962545

LNCS Sublibrary: SL4 – Security and Cryptology

Preface

The Conference on Security, Privacy, and Applied Cryptography Engineering 2018 (SPACE 2018), was held during December 15–19, 2018, at the Indian Institute of Technology Kanpur, India. This annual event is devoted to various aspects of security, privacy, applied cryptography, and cryptographic engineering. This is a challenging field, requiring expertise from diverse domains, ranging from mathematics to solid-state circuit design.

This year we received 34 submissions from 11 different countries. The submissions were evaluated based on their significance, novelty, technical quality, and relevance to the SPACE conference. The submissions were reviewed in a double-blind mode by at least three members of the 36-member Program Committee. The Program Committee was aided by 22 additional reviewers. The Program Committee meetings were held electronically, with intensive discussions. After an extensive review process, 12 papers were accepted for presentation at the conference, for an acceptance rate of 35.29%.

The program also included six invited talks and five tutorials on several aspects of applied cryptology, delivered by world-renowned researchers: Nasour Bagheri, Shivam Bhasin, Jo Van Bulck, Shay Gueron, Avi Mendelson, Mridul Nandi, Abhik Roychoudhury, Sandeep Shukla, Vanessa Teague, and Eran Toch. We sincerely thank the invited speakers for accepting our invitations in spite of their busy schedules. Like its previous editions, SPACE 2018 was organized in co-operation with the International Association for Cryptologic Research (IACR). We are thankful to the Indian Institute of Technology Kanpur for being the gracious host of SPACE 2018.

There is a long list of volunteers who invested their time and energy to put together the conference, and who deserve accolades for their efforts. We are grateful to all the members of the Program Committee and the additional reviewers for all their hard work in the evaluation of the submitted papers. We thank Cool Press Ltd., owner of the EasyChair conference management system, for allowing us to use it for SPACE 2018, which was a great help. We thank our publisher Springer for agreeing to continue to publish the SPACE proceedings as a volume in the *Lecture Notes in Computer Science* (LNCS) series. We are grateful to the local Organizing Committee, especially to the organizing chair, Sandeep Shukla, who invested a lot of effort for the conference to run smoothly. Our sincere gratitude to Debdeep Mukhopadhyay, Veezhinathan Kamakoti, and Sanjay Burman for being constantly involved in SPACE since its very inception and responsible for SPACE reaching its current status.

Last, but certainly not least, our sincere thanks go to all the authors who submitted papers to SPACE 2018, and to all the attendees. The conference is made possible by you, and it is dedicated to you. We sincerely hope you find the proceedings stimulating and inspiring.

October 2018

Anupam Chattopadhyay
Chester Rebeiro
Yuval Yarom

Organization

General Co-chairs

Sandeep Shukla	Indian Institute of Technology Kanpur, India
Manindra Agrawal	Indian Institute of Technology Kanpur, India

Program Co-chairs

Anupam Chattopadhyay	Nanyang Technological University, Singapore
Chester Rebeiro	Indian Institute of Technology Madras, India
Yuval Yarom	The University of Adelaide, Australia

Local Organizing Committee

Biswabandan Panda	Indian Institute of Technology Kanpur, India
Pramod Subramanyan	Indian Institute of Technology Kanpur, India
Shashank Singh	Indian Institute of Technology Kanpur, India

Young Researcher's Forum

Santanu Sarkar	Indian Institute of Technology Madras, India
Vishal Saraswat	Indian Institute of Technology Jammu, India

Web and Publicity

Sourav Sen Gupta	Nanyang Technological University, Singapore

Program Committee

Divesh Aggarwal	Ecole Polytechnique Fédérale de Lausanne, France
Reza Azarderakhsh	Florida Atlantic University, USA
Lejla Batina	Radboud University, The Netherlands
Shivam Bhasin	Temasek Labs, Singapore
Swarup Bhunia	University of Florida, USA
Billy Brumley	Tampere University of Technology, Finland
Arun Balaji Buduru	Indraprastha Institute of Information Technology Delhi, India
Claude Carlet	University of Paris 8, France
Rajat Subhra Chakraborty	Indian Institute of Technology Kharagpur, India
Anupam Chattopadhyay	Nanyang Technological University, Singapore
Jean-Luc Danger	Institut Télécom/Télécom ParisTech, CNRS/LTCI, France

Thomas De Cnudde	K.U. Leuven, Belgium
Junfeng Fan	Open Security Research, China
Daniel Gruss	Graz University of Technology, Austria
Sylvain Guilley	Institut Télécom/Télécom ParisTech, CNRS/LTCI, France
Jian Guo	Nanyang Technological University, Singapore
Naofumi Homma	Tohoku University, Japan
Kwok Yan Lam	Nanyang Technological University, Singapore
Yang Liu	Nanyang Technological University, Singapore
Subhamoy Maitra	Indian Statistical Institute Kolkata, India
Mitsuru Matsui	Mitsubishi Electric, Japan
Philippe Maurine	LIRMM, France
Bodhisatwa Mazumdar	Indian Institute of Technology Indore, India
Pratyay Mukherjee	Visa Research, USA
Debdeep Mukhopadhyay	Indian Institute of Technology Kharagpur, India
Chester Rebeiro	Indian Institute of Technology Madras, India
Bimal Roy	Indian Statistical Institute, Kolkata, India
Somitra Sanadhya	Indian Institute of Technology Ropar, India
Vishal Saraswat	Indian Institute of Technology Jammu, India
Santanu Sarkar	Indian Institute of Technology Madras, India
Sourav Sengupta	Nanyang Technological University, Singapore
Sandeep Shukla	Indian Institute of Technology Kanpur, India
Sujoy Sinha Roy	KU Leuven, Belgium
Mostafa Taha	Western University, Canada
Yuval Yarom	The University of Adelaide, Australia
Amr Youssef	Concordia University, Canada

Additional Reviewers

Cabrera Aldaya, Alejandro
Carre, Sebastien
Chattopadhyay, Nandish
Chauhan, Amit Kumar
Datta, Nilanjan
Guilley, Sylvain
Hou, Xiaolu
Jap, Dirmanto
Jha, Sonu
Jhawar, Mahavir
Kairallah, Mustafa
Marion, Damien
Massolino, Pedro Maat
Mozaffari Kermani, Mehran
Méaux, Pierrick
Patranabis, Sikhar
Poll, Erik
Raikwar, Mayank
Roy, Debapriya Basu
Saarinen, Markku-Juhani Olavi
Saha, Sayandeep

Keynote Talks/Tutorials Talks

Symbolic Execution vs. Search for Software Vulnerability Detection and Patching

Abhik Roychoudhury

School of Computing, National University of Singapore
abhik@comp.nus.edu.sg

Abstract. Many of the problems of software security involve search in a large domain, for which biased random searches have been traditionally employed. In the past decade, symbolic execution via systematic program analysis has emerged as a viable alternative to solve these problems, albeit with higher overheads of constraint accumulation and back-end constraint solving. We take a look at how some of the systematic aspect of symbolic execution can be imparted into biased-random searches. Furthermore, we also study how symbolic execution can be useful for purposes other than guided search, such as extracting the intended behavior of a buggy/vulnerable application. Extracting the intended program behavior, enables software security tasks such as automated program patching, since the intended program behavior can provide the correctness criterion for guiding automated program repair.

Keywords: Fuzz testing · Grey-box fuzzing · Automated program repair

1 Introduction

Software security typically involves a host of problems ranging from vulnerability detection, exploit generation, reaching nooks and corners of software for greater coverage, program hardening and program patching. Many of these problems can be envisioned as huge search problems, for example the problem of vulnerability detection can be seen as a search for problematic inputs in the input space. Similarly the problem of repairing or healing programs automatically can be seen as searching in the (huge) space of candidate patches or mutations. For these reasons, biased random searches have been used for many search problems in software security. In these settings, a more-or-less random search is conducted over a domain with the search being guided or biased by an objective function. The migration from one part of the space to another is aided by some mutation operators. A common embodiment of such biased random searches is the genetic search inherent in popular grey-box fuzzers like American Fuzzy Lop or AFL [1] which try to find inputs to crash a given program.

In the past decade symbolic or concolic execution has emerged as a viable alternative for guiding huge search problems in software security. Roughly speaking, symbolic execution works in one of two modes. Either the program is executed with a symbolic or unknown input and an execution tree is constructed. Then, the constraint along each root-to-leaf path in the tree is solved to generate sample inputs or tests.

Alternatively, in concolic execution, a random input i is generated and the constraint capturing its execution path is constructed to capture all inputs which follow the same path as i. Subsequently, the constraint captured from i's path is systematically mutated and the mutated constraints are solved to find inputs traversing other paths in the program. The aim is to enhance the path coverage for the set of inputs generated. The path constraint for a program path π, denoted $pc(\pi)$ captures the set of inputs which trace the path π.

An overview of symbolic execution for vulnerability detection and test generation appears in [2].

2 Symbolic Analysis Inspired Search

Let us consider the general problem of software vulnerability detection. Symbolic execution and search techniques both have their pros and cons. For this reason, software vulnerability detection or fuzz testing considers three flavors. The goal here is to generate program inputs which will expose program vulnerabilities. Thus, it involves a search over the domain of program inputs.

– Black-box fuzzing considers the input domain and performs mutations on program inputs, without any view of the program.
– Grey-box fuzzing has a limited view of the program such as transitions between basic blocks via compile-time instrumentation. The instrumentation helps us predict during run-time about the coverage achieved by existing set of tests, and accordingly mutations can be employed on selected tests to enhance coverage.
– White-box fuzzing has a full view of the program, which is analyzed via symbolic execution. Symbolic execution along a path produces a logical formula in the form of a path constraint. The path constraint is mutated, and the mutated logical formula is solved to (repeatedly) generate tests traversing other program paths.

Symbolic execution is clearly more systematic than grey-box/black-box fuzzing, and it is geared to traverse a new program path, when a new test input is generated. At the same time, it comes with the overheads of constraint solving and program analysis. In recent work, we have studied how ideas inspired from symbolic execution can augment the underlying genetic search in a grey-box fuzzer, such as AFL. In our recent work on AFLFast [3], we have suggested a prioritization mechanism for mutating inputs. In conventional AFL, any input selected for mutation is treated "similarly", that is, any selected input may be mutated a fixed number of times to examine the"neighbourhood" of the input. Instead, given an input, we seek to predict whether the input traces a "rare" path, a path that is frequented by few inputs. For these predicted rare paths, we subject them to enhanced examination by mutating such inputs more number of times. The amount of mutation done for a test input is governed by a so-called *power schedule*.

Another use of symbolic execution lies in reachability analysis. Specifically it is useful for finding the constraints under which a location can be reached. If paths π_1 and π_2 reach a control location L in the program, then inputs reaching L can be obtained by solving $pc(\pi_1) \vee pc(\pi_2)$ where $pc(\pi_i)$ is the path constraint for path π_i. We can incorporate this kind of reachability analysis into the genetic search inherent in grey-box fuzz testing tools like AFL. In a recent work, we have developed AFLGo [4], a directed greybox fuzzer built on top of AFL [1]. Given one or more target locations to reach, at the compile time, we instrument each basic block with approximate values of distance to the target location(s). The distance is then used in the power schedule. At the initial stages of a fuzzing session, thus the distance is not used and the search is primarily geared towards exploration. At a certain point of time, the search moves from exploration to exploitation and tries to devote more time mutating inputs whose paths are deemed to be close to the target location(s). Such an enhancement of grey-box fuzzing is an example of how the systematic nature of symbolic analysis can be imparted into search-based software security tasks.

3 Symbolic Reasoning for Program Repair

Of late, we have also explored how symbolic reasoning can be used for purposes other than guiding search or reaching locations in a program. In particular, we observe that symbolic execution can be used to extract a specification of the intended behavior of a program, directly by analyzing a buggy program. This, indeed, is a key issue, since formal specifications of intended behavior are often not available. As a result, we can envision using symbolic execution for completely new purposes, such as automated program repair or self-healing software.

The problem of automated program repair can be formally stated as follows. Given a buggy program P and a correctness criterion given as a set of tests T, how do we construct P', the minimal modification of P which passes the test-suite T.

Once again, the problem of program repair can be seen as a huge search problem in itself, it involves searching in the huge space of candidate patches of P. For this purpose, genetic search has been employed as evidenced in the GenProg tool [5]. Such a tool is based on a generate and validate approach, patches are generated, often by copying/mutating code from elsewhere in the program or from earlier program versions, and these generated patches are checked against the given tests T. Genetic search has also been used for program transplantation, a problem related to program repair, where key functionality is transplanted from one program to another [6].

Given certain weak specifications of correctness, such as a given test-suite T, we can instead try to extract a glimpse of the specification about intended program behavior, using symbolic execution. Such specifications can act as a *repair constraint*, a constraint that needs to be satisfied for the program to pass T. Subsequently, program synthesis technology can be used to synthesize patches meeting the repair constraint. Such an approach has been suggested by the SemFix work [7] and subsequently made more scalable via the Angelix tool [8] which performs multi-line program repair. Furthermore, such general purpose program repair tools have been shown to be useful for automatically generating patches for well-known security vulnerabilities such as the Heartbleed vulnerability.

There also exist opportunities for generating patches systematically from earlier program versions if one is available. If an earlier program version is available, one can repair for absence of regressions via the simultaneous symbolic analysis of the earlier and current program versions [9]. Such a technique leads to *provably correct repairs*, which can greatly help in making automated program repair an useful tool in building trustworthy systems.

Acknowledgments. This research is supported in part by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate.

References

1. Zalewski, M.: American fuzzy lop (2018). http://lcamtuf.coredump.cx/afl/
2. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
3. Böhme, M., Van Pham, T., Roychoudhury, A.: Coverage based greybox fuzzing as a markov chain. In: 23rd ACM Conference on Computer and Communications Security (CCS) (2016)
4. Böhme, M., Pham, V.-T., Nguyen, M.-D., Roychoudhury, A.: Directed greybox fuzzing. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS) (2017)
5. Weimer, W., Nguyen, T.V., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: ACM/IEEE International Conference on Software Engineering (ICSE) (2009)
6. Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F.: The plastic surgery hypothesis. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE) (2014)
7. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In: ACM/IEEE International Conference on Software Engineering (ICSE) (2013)
8. Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: ACM/IEEE International Conference on Software Engineering (ICSE) (2016)
9. Mechtaev, S., Nguyen, M.-D., Noller, Y., Lars, G., Roychoudhury, A.: Semantic program repair using a reference implementation. In: ACM/IEEE International Conference on Software Engineering (ICSE) (2018)

Persistence Wears down Resistance: Persistent Fault Analysis on Unprotected and Protected Block Cipher Implementations (Extended Abstract)

Shivam Bhasin[1], Jingyu Pan[1,2], and Fan Zhang[2]

[1] Temasek Laboratories, Nanyang Technological University, Singapore
sbhasin@ntu.edu.sg
[2] Zhejiang University, China
joeypan,fanzhangg@zju.edu.cn

Abstract. This works gives an overview of persistent fault attacks on block ciphers, a recently introduced fault analysis technique based on persistent faults. The fault typically targets stored constant of cryptographic algorithms over several encryption calls with a single injection. The underlying analysis technique statistically recovers the secret key and is capable of defeating several popular countermeasures by design.

Keywords: Fault attacks · Modular redundancy · Persistent fault

1 Introduction

Fault attacks [1, 2] are active physical attacks that use external means to disturb normal operations of a target device leading to security vulnerability. These attacks have been widely used for key recovery from widely used standard cryptographic schemes, such as AES, RSA, ECC etc.

Several types of faults can be exploited to mount such attacks. Commonly known fault types are *transient* and *permanent*. A transient fault, which is most commonly used, generally affects only one instance of the target function call (eg. one encryption). On the other hand, a permanent fault, normally owing to device defects, affects all calls to the target function. Based on these two fault types, several analysis techniques have been developed. The most common are differential in nature, which require a correct and faulty computation with same inputs, to exploit the difference of final correct and faulty output pair for key recovery. Common examples of such techniques are differential fault analysis (DFA) [2], algebraic fault analysis (AFA) [4], etc. Some analyses are also based on statistical methods which can exploit faulty ciphertexts only like statistical fault analysis (SFA) [5] and fault sensitivity analysis (FSA) [6].

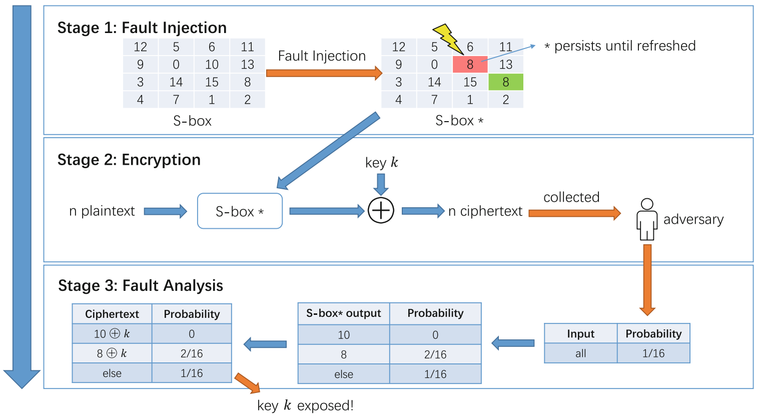
Recently, a new fault analysis technique was proposed [8] with a *persistent* fault model. Persistent fault lies between transient and permanent faults. Unlike transient fault, it affects several calls of the target function, however, persistent fault is not permanent, and disappears with a device reset/reboot. The corresponding analysis technique is called *Persistent Fault Analysis (PFA)* [8].

2 Persistent Fault Analysis (PFA)

PFA [8] is based on persistent fault model. In the following, the fault is assumed to alter a stored constant (like one or more entries in Sbox look-up) in the target algorithm, typically stored in a ROM. The attacker observes multiple ciphertext outputs with varying plaintext (not known). The modus operandi of PFA is explained with the following example. Let us take PRESENT cipher which uses a 4×4 Sbox i.e. 16 elements of 4-bits each, where each element has an equal expectation $\mathbb{E}$ of $\frac{1}{16}$. A persistent fault alters value of element x in Sbox to another element x', it makes $\mathbb{E}(x) = 0, \mathbb{E}(x') = \frac{2}{16}$, while all other elements still have the expectation $\frac{1}{16}$. The output ciphertext is still correct if faulty element x is never accessed during the encryption else the output is faulty. This difference can be statistically observed in the final ciphertext where some values will be missing (related to x) and some occuring more often than others (due to x'), which leaks information on the key k. This is illustrated in Fig. 1 (top) with $x = 10, x' = 8$. The key can be recovered even if x, x' are not known by simple brute-forcing. The strategy for key recovery can be one of the following:

1. t_{min}: find the missing value in Sbox table. Then $k = t_{min} \oplus x$;
2. $t \neq t_{min}$: find other values t where $t \neq t_{min}$ and eliminate candidates for k;
3. t_{max}: find the value with maximal probability $k = t_{max} \oplus x'$.

The distribution of t_{min} or t_{max} can be statistically distinguished from the rest. The minimum number of ciphertexts N follows the classical coupon collector's problem [3] where it needs $N = (2^b - 1) \times (\sum_{i=1}^{(2^b-1)} \frac{1}{i})$, where b is the bit width of x. In PRESENT ($b = 4$) $N \approx 50$, as shown in Fig. 1 (bottom).



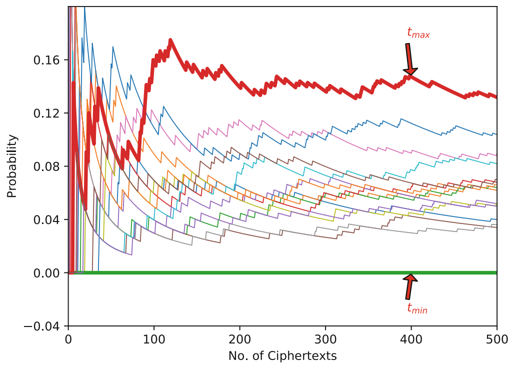


Fig. 1. Overview of Persistent Fault Attack (top), distribution of t_{min} and t_{max} against no. of ciphertexts for PRESENT leading to key recovery (bottom)

2.1 PFA vs Other Fault Analysis

Here we list the key merits and demerits of PFA against other fault analysis.

Merits

– The main advantage of PFA is that it needs only one fault injection, which reduces the injection effort to minimum. Fault targets a constant in memory which persists over several following encryptions. This also reduces the injection effort in terms of timing precision within an injection. Moreover, live detection by sensors can be bypassed by injecting before the sensitive computation starts and sensors become active.
– The attack is statistical on ciphertexts only, and thus unlike differential attacks, needs no control over plaintexts.

- The fault model remains relaxed compared to other statistical attacks which may require multiple injections (one per encryption) with a known bias or additional side-channel information.
- Unlike any other known attacks, PFA can also be applied in the multiple fault (in a single encryption) setting.

Demerits

- Being statistical in nature, it needs a higher number of ciphertexts as compared to DFA. Some known DFA can lead to key recovery with 1 or 2 correct/faulty ciphertext pair.
- Persistent faults can be detected by built-in self check mechanism.

2.2 Application of PFA on Countermeasures

PFA has natural properties which make several countermeasures vulnerable. The details on analysis of the countermeasure remain out of scope of this extended abstract due to limited space and interested readers are encouraged to refer [8]. Dual modular redundancy (DMR) is a popular fault countermeasure. The most common DMR proposes to compute twice and compare outputs. This countermeasure is naturally vulnerable to PFA if shared memories for constants are used, which is often the case due to resource constraint environments. Other proposals use separate memories or compute forward followed by compute inverse and compare inputs. All these countermeasures output a correct ciphertext when no fault is injected. For a detected fault, the faulty output can be suppressed by no ciphertext output (NCO), zero value output (ZVO), or random ciphertext output (RCO) [8]. As PFA leaves certain probability for correct ciphertext output despite the persistent fault, it leads to key recovery using statistical method. However, more ciphertexts would be required in the analysis as some information is suppressed by the DMR countermeasure. Masking [7] on the other hand is a side channel countermeasure which is widely studied. As a persistent fault injects a bias in the underlying computation due to a biased constant, the bias can also affect the masking leading to key recovery.

3 Conclusion

Persistent fault analysis is a powerful attack technique which can make several cryptographic schemes vulnerable. With as low as one fault injection and simple statistical analysis on ciphertexts, PFA can perform key recovery. The introduced vulnerability also extends to protected implementations. We briefly discussed the impact of PFA on modular redundancy and masking based countermeasures. Existing countermeasures and other cryptographic schemes including public key cryptography must be analyzed to check their resistance against PFA. This further motivates research for dedicated countermeasures to prevent PFA.

References

1. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer's apprentice guide to fault attacks. Proc. IEEE **94**(2), 370–382 (2006)
2. Biham, E., Shamir, A.: Differential cryptanalysis of the data encryption standard. Cryst. Res. Technol. **17**(1), 79–89 (2006)
3. Blom, G., Holst, L., Sandell, D.: Problems and Snapshots from the World of Probability. Springer, Heidelberg (2012)
4. Courtois, N.T., Jackson, K., Ware, D.: Fault-algebraic attacks on inner rounds of DES. In: e-Smart'10 Proceedings: The Future of Digital Security Technologies. Strategies Telecom and Multimedia (2010)
5. Fuhr, T., Jaulmes, E., Lomne, V., Thillard, A.: Fault attacks on AES with faulty ciphertexts only. In: The Workshop on Fault Diagnosis & Tolerance in Cryptography, pp. 108–118 (2013)
6. Li, Y., Sakiyama, K., Gomisawa, S., Fukunaga, T., Takahashi, J., Ohta, K.: Fault sensitivity analysis. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 320–334. Springer, Heidelberg (2010)
7. Rivain, M., Prouff, E.: Provably secure higher-order masking of AES. CHES **2010**, 413–427 (2010)
8. Zhang, F., Lou, X., Zhao, X., Shivam, B., He, W., Ding, R., Qureshi, S., Ren, K.: Persistent fault analysis on block ciphers. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(3), 150–172 (2018)

Tutorial: Uncovering and Mitigating Side-Channel Leakage in Intel SGX Enclaves

Jo Van Bulck and Frank Piessens

imec-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium
{jo.vanbulck,frank.piessens}@cs.kuleuven.be

Abstract. The inclusion of the Software Guard eXtensions (SGX) in recent Intel processors has been broadly acclaimed for bringing strong hardware-enforced trusted computing guarantees to mass consumer devices, and for protecting end user data in an untrusted cloud environment. While SGX assumes a very strong attacker model and indeed even safeguards enclave secrets against a compromised operating system, recent research has demonstrated that considerable private data (e.g., full text and images, complete cryptographic keys) may still be reconstructed by monitoring subtle side-effects of the enclaved execution. We argue that a systematic understanding of such side-channel leakage sources is essential for writing intrinsically secure enclave applications, and will be instrumental to the success of this new trusted execution technology. This tutorial and write-up therefore aims to bring a better understanding of current state-of-the-art side-channel attacks and defenses on Intel SGX platforms. Participants will learn how to extract data from elementary example applications, thereby recognizing how to avoid common pitfalls and information leakage sources in enclave development.

Keywords: Side-channel · Enclave · SGX · Tutorial

1 Introduction

Trusted Execution Environments (TEEs), including Intel SGX, are a promising new technology supporting secure isolated execution of critical code in dedicated *enclaves* that are directly protected and measured by the processor itself. By excluding vast operating system and hypervisor code from the trusted computing base, TEEs establish a minimalist hardware root-of-trust where application developers solely rely on the correctness of the CPU and the implementation of their own enclaves. Enclaved execution hence holds the promise of enforcing strong security and privacy requirements for local and remote computations.

Modern processors unintendedly leak information about (enclaved) software running on top, however, and such traces in the microarchitectural CPU state can be abused to reconstruct application secrets through side-channel analysis. These attacks have received growing attention from the research community and significant understanding has been built up over the past decade. While information leakage from side-channels is generally limited to specific code or data access patterns, recent work

[4, 5, 8–11] has demonstrated significant side-channel amplification for enclaved execution. Ultimately, the disruptive real-world impact of side-channels became apparent when they were used as building blocks for the high-impact Meltdown, Spectre, and Foreshadow speculation attacks (where the latter completely erodes trust on unpatched Intel SGX platforms [7]).

Intel explicitly considers side-channels out of scope, clarifying that “it is the enclave developer’s responsibility to address side-channel attack concerns” [2]. Unfortunately, we will show that adequately preventing side-channel leakage is particularly difficult — to the extent where even Intel’s own vetted enclave entry code suffered from subtle yet dangerous side-channel vulnerabilities [3]. As such, we argue that side-channels cannot merely be considered out of scope for enclaved execution, but rather necessitate widespread developer education so as to establish a systematic understanding and awareness of different leakage sources. To support this cause, this tutorial and write-up present a brief systematization of current state-of-the-art attacks and general guidelines for secure enclave development.

All presentation material and source code for this tutorial will be made publicly available at https://github.com/jovanbulck/sgx-tutorial-space18.

2 Software Side-Channel Attacks on Enclaved Execution

We consider a powerful class of software-only attacks that require only code execution on the machine executing the victim enclave. Depending on the adversary’s goals and capabilities, the malicious code can either be executing interleaved with the victim enclave (interrupt-driven attacks [4, 8–11]), or launched concurrently from a co-resident logical CPU core (HyperThreading-based resource contention attacks [5]). In the following, we overview known side-channels.

Memory Accesses. Even before the official launch of Intel SGX, researchers showed the existence of a dangerous side-channel [11] within the processor’s virtual-to-physical address translation logic. By revoking access rights on selected enclave memory pages, and observing the associated page fault patterns, adversaries controlling the operating system can deterministically establish enclaved code and data accesses at a 4 KiB granularity. This attack technique has been proven highly practical and effective, extracting full enclave secrets in a single run and without noise. Following the classic cat-and-mouse game, subsequent proposals to hide enclave page faults from the adversary led to an improved class of stealthy attack variants [10] that extract page table access patterns without provoking page faults. It has furthermore been demonstrated [8] that privileged adversaries can mount such interrupt-driven attacks at a very precise instruction-level granularity, which allows to accurately monitor enclave memory access patterns in the time domain so as to defeat naive spatial page alignment defense techniques [2, 8].

A complementary line of SGX-based Prime+Probe cache attacks exploit information leakage at an improved 64-byte cache line granularity [6]. Adversaries first load carefully selected memory locations into the shared CPU cache, and afterwards measure the time to reload these addresses to establish code and data evictions by the victim enclave. As with the paging channel above, these attacks commonly exploit the adversary's control over untrusted system software to frequently interrupt the victim enclave and gather side-channel information at a maximum temporal resolution [8]. This is not a strict requirement, however, as it has been demonstrated that even unprivileged attacker processes can concurrently monitor enclave cache access patterns in real-time [6].

In summary, the above research results show that enclave code and data accesses on SGX platforms can be accurately reconstructed, both in space (at a 4 KiB or 64-byte granularity) as well as in time (after every single instruction).

Instruction-Level Leakage. It has furthermore been shown that enclave-private control flow leaks through the CPU's internal Branch Target Buffer [4]. These attacks essentially follow the general principle of the above Prime+Probe attacks by first forcing the BTB cache in a known state. After interrupting the enclave, the adversary measures a dedicated shadow branch to establish whether the secret-dependent victim branch was executed or not. Importantly, unlike the above memory access side-channels, such branch shadowing attacks leak control flow at the level of individual branch instructions (i.e., basic blocks).

Apart from amplifying conventional side-channels, enclaved execution attack research has also revealed new and unexpected sub-cache level leakage sources. One recent work presented the Nemesis [9] attack that measures individual enclaved instruction timings through interrupt latency, allowing to partially reconstruct a.o., instruction type, operand values, address translation, and cache hits/misses. MemJam [5] furthermore exploits selective instruction timing penalties from false dependencies induced by an attacker-controlled spy thread to reconstruct enclave-private memory access patterns *within* a 64-byte cache line.

Speculative Execution. In the aftermath of recent x86 speculation vulnerabilities, researchers have successfully demonstrated Spectre-type speculative code gadget abuse against SGX enclaves [1]. Recent work furthermore presented Foreshadow [7] which allows for arbitrary in-enclave reads and completely dismantles isolation and attestation guarantees in the SGX ecosystem. Intel has since revoked the compromised attestation keys, and released microcode patches to address Foreshadow and Spectre threats at the hardware level.

3 Enclave Development Guidelines and Caveats

Existing SGX side-channel mitigation approaches generally fall down in two categories. One line of work attempts to harden enclave programs through a combination of compile time code rewriting and run time randomization or checks, so as to obfuscate the attacker's view or detect side-effects of an ongoing attack. Unfortunately, as these heuristic proposals do not block the root information leakage in itself, they often fall victim to improved and more versatile attack variants [5, 8, 10]. A complementary line of work therefore advocates the more comprehensive constant time approach known from the cryptography community: eliminate secret-dependent code and data paths altogether. While this approach is relatively well-understood for small applications, in practice even vetted crypto implementations exhibit non-constant time behavior [5, 6, 10]. In the context of SGX, it has furthermore been shown [9, 11] that enclave secrets are typically not limited to well-defined private keys, but are instead scattered throughout the code and hence much harder to manipulate in constant time.

We conclude that side-channels pose a real threat to enclaved execution, while no silver bullet exists to eliminate them at the compiler or system level. Depending on the enclave's size and security objectives, it may be desirable to strive for intricate constant time solutions, or instead rely on heuristic hardening measures. However, further research and raising developer awareness are imperative to make such informed decisions and adequately employ TEE technology.

Acknowledgments. This research is partially funded by the Research Fund KU Leuven. Jo Van Bulck is supported by the Research Foundation – Flanders (FWO).

References

1. Chen, G., Chen, S., Xiao, Y., Zhang, Y., Lin, Z., Lai, T.H.: SgxPectre attacks: Leaking enclave secrets via speculative execution arXiv:1802.09085 (2018)
2. Intel: Software guard extensions developer guide: Protection from side-channel attacks (2017). https://software.intel.com/en-us/node/703016
3. Intel: Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Edger8r Generated Code Side Channel Exploits (2018)
4. Lee, S., Shih, M.W., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In: Proceedings of the 26th USENIX Security Symposium, Vancouver, Canada (2017)
5. Moghimi, A., Eisenbarth, T., Sunar, B.: Memjam: a false dependency attack against constant-time crypto implementations in SGX. In: Smart, N.P. (ed.) Cryptographers' Track at the RSA Conference, LNCS, vol 10808, pp. 21–44. Springer, Cham (2018)
6. Schwarz, M., Weiser, S., Gruss, D., Maurice, C., Mangard, S.: Malware guard extension: using SGX to conceal cache attacks. In: Polychronakis M., Meier M. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. DIMVA 2017. LNCS, vol 10327, pp. 3–24. Springer, Cham (2017)
7. Van Bulck, J., et al.: Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: Proceedings of the 27th USENIX Security Symposium. USENIX Association (2018)
8. Van Bulck, J., Piessens, F., Strackx, R.: SGX-Step: a practical attack framework for precise enclave execution control. In: Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX'17, pp. 4:1–4:6. ACM (2017)
9. Van Bulck, J., Piessens, F., Strackx, R.: Nemesis: studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In: Proceedings of the 25th ACM Conference on Computer and Communications Security, CCS'18 (2018)

10. Van Bulck, J., Weichbrodt, N., Kapitza, R., Piessens, F., Strackx, R.: Telling your secrets without page faults: stealthy page table-based attacks on enclaved execution. In: Proceedings of the 26th USENIX Security Symposium, USENIX (2017)
11. Xu, Y., Cui, W., Peinado, M.: Controlled-channel attacks: deterministic side channels for untrusted operating systems. In: 2015 IEEE Symposium on Security and Privacy, pp. 640–656. IEEE (2015)

A Composition Result for Constructing BBB Secure PRF

Nilanjan Datta[1], Avijit Dutta[2], Mridul Nandi[2], Goutam Paul[2]

[1] Indian Institute of Technology, Kharagpur
[2] Indian Statistical Institute, Kolkata
nilanjan_isi_jrf@yahoo.com,
avirocks.dutta13@gmail.com, mridul.nandi@gmail.com,
goutam.paul@isical.ac.in

Abstract. In this paper, we propose **Double-block Hash-then-Sum** (DbHtS), a generic design paradigm to construct a BBB secure pseudo random function. DbHtS computes a *double block hash* function on the message and then *sum* the encrypted outputs of the two hash blocks. Our result renders that if the underlying hash function meets certain security requirements (namely cover-free and block-wise universal advantage is low), DbHtS construction provides $2n/3$-bit security. We demonstrate the applicability of our result by instantiating all the existing beyond birthday secure deterministic MACs (e.g., SUM-ECBC, PMAC_Plus, 3kf9, LightMAC_Plus) as well as their reduced-key variants.

1 Introduction

Pseudo Random Function (PRF) plays an important role in symmetric key cryptography to authenticate or encrypt any arbitrary length message. Over the years, there have numerous candidates of PRFs (e.g., CBC-MAC [BKR00] and many others). These PRFs give security only upto *birthday bound*, i.e., the mode is secure only when the total number of blocks that the mode can process does not exceed $2^{n/2}$, where n is the block size of the underlying primitive (e.g., block cipher). Birthday bound secure constructions are acceptable in practice with a moderately large block size. However, the mode becomes vulnerable if instantiated with some smaller block size primitive. In this line of research, SUM-ECBC [Yas10] is the first beyond the birthday bound (BBB) secure rate-1 / 2 PRF with $2n/3$-bit security. Followed by this work, many BBB secure PRFs e.g., PMAC_Plus [Yas11], 3kf9 [ZWSW12], LightMAC_Plus [Nai17], 1K-PMAC_Plus [DDN+17] etc. have been proposed and all of them gives roughly $2n/3$-bit security. Interestingly, all these constructions possess a common structural design which is the composition of (i) a double block hash (DbH) function that outputs a $2n$-bit hash value of the input message and (ii) a finalization phase that generates the final tag by xor-ing the encryption of two n-bit hash values. However, all these PRFs follow a different way to bound the security. This observation motivates us to come up with a generic design guideline to construct BBB secure PRFs and hence brings all the existing BBB secure PRFs under one common roof and enables us to give a unified security proof for all of them.

Our Contributions. We introduce **Double-block Hash-then-Sum** (DbHtS) a generic design of BBB secure PRF by xor-ing the encryption of the outputs of a DbH function. Based on the usage of the keys, we call the DbHtS construction three-keyed (resp. two-keyed), if two block cipher keys are (resp. a single block cipher key is) used in the finalization phase along with the hash key. We show that if the cover-free and the block-wise universal advantage of the underlying DbH function is sufficiently low, then the two-keyed DbHtS is secure beyond the birthday bound. We show the applicability of this result by instantiating existing beyond birthday secure deterministic MACs (i.e., SUM-ECBC, PMAC_Plus, 3kf9, LightMAC_Plus) and their two-keyed variants and showing their beyond birthday bound security.

2 DbHtS : A BBB Secure PRF Paradigm

Double-block Hash-then-Sum (DbHtS) is a paradigm to build a BBB secure VIL PRF where the Double-block Hash (DbH) function is used with a very simple and efficient single-keyed or two-keyed sum function:

- Single-Keyed Sum Function: $Sum_K(x, y) = E_K(x) \oplus E_K(y)$,
- Two-Keyed Sum Function: $Sum_{K_1,K_2}(x, y) = E_{K_1}(x) \oplus E_{K_2}(y)$.

Given a DbH function and the sum function over two blocks, we apply the composition of the DbH function and the sum function to realize the DbHtS construction. Based on the types of sum function i.e., single-keyed or (resp. two-keyed) used in the composition, we have three keyed or (resp. two-keyed) DbHtS construction.

2.1 Security Definitions for DbH Function

Let $\mathcal{K}_b$ be a set of bad hash keys. A DbH function is said to be **(weak) cover-free** if for any triplet of messages out of any q distinct messages, the joint probability, over a random draw of the hash key, that the values taken by the two halves of the hash outputs of a message also appears in the (corresponding) either halves of the hash outputs of two other messages of the triplet, and the sampled hash key falls to the set $\mathcal{K}_b$, is low. A DbH function is said to be **(weak) block-wise universal** if for any pair of messages out of any q distinct messages, the joint probability, over a random draw of the hash key, that any half of the hash output for a message collides with (the same) any half of the hash output for the other message of the pair, and the sampled hash key falls to the set $\mathcal{K}_b$, is low. Finally, a DbH function is said to be **colliding** if for any messages out of any q distinct messages, the joint probability, over a random draw of the hash key, that any half of the hash output of the message collides with other halves of the output, and the sampled hash key falls to the set $\mathcal{K}_b$, is low.

2.2 Security Result of DbHtS

Let q denotes the maximum number of queries by any adversary and ℓ denotes the maximum number of message blocks among all q queried messages.

Theorem 1
(i) If H is a ϵ_{cf}-cover-free, ϵ_{univ}-block-wise universal and ϵ_{coll}-colliding hash function for a fixed set of bad hash keys $\mathcal{K}_b$, then the distinguishing advantage of two-keyed DbHtS from the random function is bounded by

$$\epsilon_{\text{bh}} + q\epsilon_{\text{coll}} + \frac{q^3}{6}\epsilon_{\text{cf}} + \frac{3q^3}{2^n}\epsilon_{\text{univ}} + \frac{6q^3}{2^{2n}} + \frac{q}{2^n},$$

where ϵ_{bh} is the upper bound on the probability of a sampled hash key falls to $\mathcal{K}_b$.
(ii) If H is a ϵ_{wcf}-weak cover-free and ϵ_{wuniv}-weak block-wise universal hash function for a fixed set of bad hash keys $\mathcal{K}_b$, then the distinguishing advantage of three-keyed DbHtS from the random function is bounded by

$$\epsilon_{\text{bh}} + \frac{q^3}{6}\epsilon_{\text{wcf}} + \frac{3q^3}{2^n}\epsilon_{\text{wuniv}} + \frac{2q^3}{2^{2n}},$$

where ϵ_{bh} is the upper bound on the probability of a sampled hash key falls to $\mathcal{K}_b$.

Instantiations of DbHtS

In this section, we revisit two BBB secure parallel mode PRF PMAC_Plus, LightMAC_Plus and two BBB secure parallel mode PRF SUM-ECBC, 3kf9. We also consider simple two-key variants of these constructions. All the specifications are given in Fig. 1. Applying Theorem 1 on these constructions, we obtain the following bounds:

Constructions	Security Bound	Constructions	Security Bound
2K-PMAC_Plus	$q^3\ell/2^{2n} + q^2\ell^2/2^{2n}$	2K-SUM-ECBC	$2q\ell^2/2^n + q^3\ell^2/2^{2n}$
2K-LightMAC_Plus	$q^3/2^{2n} + q/2^n$	2Kf9	$q^3\ell^4/2^{2n}$
PMAC_Plus	$q^3\ell/2^{2n} + q^2\ell^2/2^{2n}$	SUM-ECBC	$q\ell^2/2^n + q^3/2^{2n}$
LightMAC_Plus	$q^3/2^{2n}$	3kf9	$q^3\ell^4/2^{2n}$

Open Problems. Here we list down some possible future research works:

(i) One may try to extend this work to analyze the security of the single-keyed DbHtS, where the hash key would be same as the block cipher key used in the sum function.
(ii) Leurent et al [LNS18] have shown attacks on SUM-ECBC, PMAC_Plus, 3kf9 and LightMAC_Plus with query complexity of $O(2^{3n/4})$. Establishing the tightness of the bound is an interesting open problem.

```
Algorithm $\boxed{\mathsf{2K}}$-PMAC_Plus$(M)$

1. $\Delta_0 \leftarrow E_K(0^n);\ \Delta_1 \leftarrow E_K(0^{n-1}1)$
2. for $j = 1$ to $l$
3.     $X_j = M_j \oplus 2^j \Delta_0 \oplus 2^{2j} \Delta_1$
4.     $Y_j = E_K(X_j)$
5. $\Sigma' = \oplus_{i=1}^{l} Y_i; \Lambda' = \oplus_{i=1}^{l} 2^{l-i} \cdot Y_i$
6. $\boxed{\Sigma = \mathsf{fix0}(\Sigma');\ \Lambda = \mathsf{fix1}(2\Lambda')}$
7. return $E_{K_1}(\Sigma') \oplus E_{K_2}(\Lambda')$
8. return $\boxed{E_K(\Sigma) \oplus E_K(\Lambda)}$
```

```
Algorithm $\boxed{\mathsf{2K}}$-LightMAC_Plus$(M)$

1. for $j = 1$ to $l$
2.     $X_j = \langle j \rangle_s \| M_j$
3.     $Y_j = E_K(X_j)$
4. $\Sigma' = \oplus_{i=1}^{l} Y_i; \Lambda' = \oplus_{i=1}^{l} 2^{l-i} \cdot Y_i$
5. $\boxed{\Sigma = \mathsf{fix0}(\Sigma');\ \Lambda = \mathsf{fix1}(2\Lambda')}$
6. return $E_{K_1}(\Sigma') \oplus E_{K_2}(\Lambda')$
7. return $\boxed{E_K(\Sigma) \oplus E_K(\Lambda)}$
```

```
Algorithm $\boxed{\mathsf{2K}}$-SUM-ECBC$(M)$

1. $(Y, Y') \leftarrow (0^n, 0^n)$
2. for $j = 1$ to $l$
3.     $X = M_j \oplus Y; X' = M_j \oplus Y'$
4.     $(Y, Y') \leftarrow (E_K(X), E_{K_\star}(X'))$
5. $(\Sigma' \Lambda') \leftarrow (Y, Y')$
6. $\boxed{\Sigma \leftarrow \mathsf{fix0}(\Sigma'); \Lambda \leftarrow \mathsf{fix1}(\Theta')}$
7. return $E_{K_1}(\Sigma') \oplus E_{K_2}(\Lambda')$
8. return $\boxed{E_K(\Sigma) \oplus E_K(\Lambda')}$
```

```
Algorithm $\boxed{\mathsf{2Kf9}}$3kf9$(M)$

1. $(Y, Y') \leftarrow (0^n, 0^n)$
2. for $j = 1$ to $l$
3.     $X = M_j \oplus Y; Y \leftarrow E_K(X)$
4.     $Y' \leftarrow Y \oplus Y'$
5. $(\Sigma', \Lambda') = (Y, Y')$
6. return $E_{K_1}(\Sigma') \oplus E_{K_2}(\Lambda')$
7. return $\boxed{E_K(\Sigma') \oplus E_K(\Lambda')}$
```

Fig. 1. Specification of existing MACs with BBB Security and their two-key variants. Here $\langle j \rangle_s$ denotes the s-bit binary representation of integer j and fixb function takes an n-bit integer and returns the integer with its least significant bit set to bit b.

References

[BKR00] Bellare, M., Kilian, J., Rogaway, P.: The security of the cipher block chaining message authentication code. J. Comput. Syst. Sci. **61**(3), 362–399 (2000)

[DDN+17] Datta, N., Dutta, A., Nandi, M., Paul, G., Zhang, L.: Single key variant of pmac_plus. IACR Trans. Symmetric Cryptol. **2017**(4), 268–305 (2017)

[LNS18] Leurent, G., Nandi, M., Sibleyras, F.: Generic attacks against beyond-birthday-bound macs, vol. 2018, p. 541 (2018)

[Nai17] Naito, Y.: Blockcipher-based macs: beyond the birthday bound without message length. Cryptology ePrint Archive, Report 2017/852 (2017)

[Yas10] Yasuda, K.: The sum of CBC macs is a secure PRF. CT-RSA **2010**, 366–381 (2010)

[Yas11] Yasuda, K.: A new variant of PMAC: beyond the birthday bound. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 596–609. Springer, Heidelberg (2011)

[ZWSW12] Zhang, L., Wu, W., Sui, H., Wang, P.: 3kf9: enhancing 3GPP-MAC beyond the birthday bound. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 296–312. Springer, Heidelberg (2012)

Contents