Shohreh Hosseinzadeh

Security and Trust in Cloud Computing and IoT through Applying Obfuscation, Diversification, and Trusted Computing Technologies

Turku Centre *for* Computer Science

TUCS Dissertations
No 258, November 2020

Security and Trust in Cloud Computing and IoT through Applying Obfuscation, Diversification, and Trusted Computing Technologies

Shohreh Hosseinzadeh

To be presented, with the permission of the Faculty of Science and Engineering of the University of Turku, for public criticism in lecture hall XXII of Agora on November 28th, 2020, at 12 noon.

University of Turku
Department of Future Technologies
20014 Turun Yliopisto Finland

2020

Supervisors

Professor Ville Leppänen
Department of Future Technologies
University of Turku
Finland

Associate Professor Seppo Virtanen
Department of Future Technologies
University of Turku
Finland

Reviewers

Professor Benoit Baudry
Division of Software and Computer System
KTH Royal Institute of Technology
Lindstedtsvägen 3, Stockholm
Sweden

Adjunct Professor Martin Gilje Jaatun
Department of Electrical
University of Stavanger
Kjell Arholms gate 41, 4036 Stavanger
Norway

Opponent

Professor Valtteri Niemi
Department of Computer Science
University of Helsinki
Pietari Kalmin katu 5, 00014 Helsinki
Finland

ISBN 978-952-12-3992-2
ISSN 1239-1883

Abstract

Cloud computing and Internet of Things (IoT) are very widely spread and commonly used technologies nowadays. The advanced services offered by cloud computing have made it a highly demanded technology.

Enterprises and businesses are more and more relying on the cloud to deliver services to their customers. The prevalent use of cloud means that more data is stored outside the organization's premises, which raises concerns about the security and privacy of the stored and processed data. This highlights the significance of effective security practices to secure the cloud infrastructure.

The number of IoT devices is growing rapidly and the technology is being employed in a wide range of sectors including smart healthcare, industry automation, and smart environments. These devices collect and exchange a great deal of information, some of which may contain critical and personal data of the users of the device. Hence, it is highly significant to protect the collected and shared data over the network; notwithstanding, the studies signify that attacks on these devices are increasing, while a high percentage of IoT devices lack proper security measures to protect the devices, the data, and the privacy of the users.

In this dissertation, we study the security of cloud computing and IoT and propose software-based security approaches supported by the hardware-based technologies to provide robust measures for enhancing the security of these environments. To achieve this goal, we use obfuscation and diversification as the potential software security techniques. Code obfuscation protects the software from malicious reverse engineering and diversification mitigates the risk of large-scale exploits. We study trusted computing and Trusted Execution Environments (TEE) as the hardware-based security solutions. Trusted Platform Module (TPM) provides security and trust through a hardware root of trust, and assures the integrity of a platform. We also study Intel SGX which is a TEE solution that guarantees the integrity and confidentiality of the code and data loaded onto its protected container, enclave.

More precisely, through obfuscation and diversification of the operating systems and APIs of the IoT devices, we secure them at the application level,

and by obfuscation and diversification of the communication protocols, we protect the communication of data between them at the network level. For securing the cloud computing, we employ obfuscation and diversification techniques for securing the cloud computing software at the client-side. For an enhanced level of security, we employ hardware-based security solutions, TPM and SGX. These solutions, in addition to security, ensure layered trust in various layers from hardware to the application.

As the result of this PhD research, this dissertation addresses a number of security risks targeting IoT and cloud computing through the delivered publications and presents a brief outlook on the future research directions.

Tiivistelmä

Pilvilaskenta ja esineiden internet ovat nykyään hyvin tavallisia ja laajasti sovellettuja tekniikkoja. Pilvilaskennan pitkälle kehittyneet palvelut ovat tehneet siitä hyvin kysytyn teknologian. Yritykset enenevässä määrin nojaavat pilviteknologiaan toteuttaessaan palveluita asiakkailleen. Vallitsevassa pilviteknologian soveltamistilanteessa yritykset ulkoistavat tietojensa käsittelyä yrityksen ulkopuolelle, minkä voidaan nähdä nostavan esiin huolia taltioitavan ja käsiteltävän tiedon turvallisuudesta ja yksityisyydestä. Tämä korostaa tehokkaiden turvallisuusratkaisujen merkitystä osana pilvi-infrastruktuurin turvaamista.

Esineiden internet -laitteiden lukumäärä on nopeasti kasvanut. Teknologiana sitä sovelletaan laajasti monilla sektoreilla, kuten älykkäässä terveydenhuollossa, teollisuusautomaatiossa ja älytiloissa. Sellaiset laitteet keräävät ja välittävät suuria määriä informaatiota, joka voi sisältää laitteiden käyttäjien kannalta kriittistä ja yksityistä tietoa. Tästä syystä johtuen on erittäin merkityksellistä suojata verkon yli kerättävää ja jaettavaa tietoa. Monet tutkimukset osoittavat esineiden internet -laitteisiin kohdistuvien tietoturvahyökkäysten määrän olevan nousussa, ja samaan aikaan suuri osuus näistä laitteista ei omaa kunnollisia teknisiä ominaisuuksia itse laitteiden tai niiden käyttäjien yksityisen tiedon suojaamiseksi.

Tässä väitöskirjassa tutkitaan pilvilaskennan sekä esineiden internetin tietoturvaa ja esitetään ohjelmistopohjaisia tietoturvalähestymistapoja turvautumalla osittain laitteistopohjaisiin teknologioihin. Esitetyt lähestymistavat tarjoavat vankkoja keinoja tietoturvallisuuden kohentamiseksi näissä konteksteissa. Tämän saavuttamiseksi työssä sovelletaan obfuskaatiota ja diversifiointia potentiaalisiana ohjelmistopohjaisina tietoturvatekniikkoina. Suoritettavan koodin obfuskointi suojaa pahantahtoiselta ohjelmiston takaisinmallinnukselta ja diversifiointi torjuu tietoturva-aukkojen laaja-alaisen hyödyntämisen riskiä. Väitöskirjatyössä tutkitaan luotettua laskentaa ja luotettavan laskennan suoritusalustoja laitteistopohjaisina tietoturvaratkaisuina. TPM (Trusted Platform Module) tarjoaa turvallisuutta ja luottamuksellisuutta rakentuen laitteistopohjaiseen luottamukseen. Pyrkimyksenä on taata suoritusalustan eheys. Työssä tutkitaan myös Intel SGX:ää yhtenä luotettavan suorituksen suoritusalustana, joka takaa suoritettavan koodin

ja datan eheyden sekä luottamuksellisuuden pohjautuen suojatun säiliön, saarekkeen, tekniseen toteutukseen.

Tarkemmin ilmaistuna työssä turvataan käyttöjärjestelmä- ja sovellusrajapintatasojen obfuskaation ja diversifioinnin kautta esineiden internet -laitteiden ohjelmistokerrosta. Soveltamalla samoja tekniikoita protokollakerrokseen, työssä suojataan laitteiden välistä tiedonvaihtoa verkkotasolla. Pilvilaskennan turvaamiseksi työssä sovelletaan obfuskaatio- ja diversifiointitekniikoita asiakaspuolen ohjelmistoratkaisuihin. Vankemman tietoturvallisuuden saavuttamiseksi työssä hyödynnetään laitteistopohjaisia TPM- ja SGX-ratkaisuja. Tietoturvallisuuden lisäksi nämä ratkaisut tarjoavat monikerroksisen luottamuksen rakentuen laitteistotasolta ohjelmistokerrokseen asti.

Tämän väitöskirjatutkimustyön tuloksena, osajulkaisuiden kautta, vastataan moniin esineiden internet -laitteisiin ja pilvilaskentaan kohdistuviin tietoturvauhkiin. Työssä esitetään myös näkemyksiä jatkotutkimusaiheista.

Acknowledgments

Undertaking this PhD has truly been a life-changing experience to me, and marks a milestone on my personal and professional life. Reaching this milestone would not have been possible without the guidance and support of many people.

First and foremost, I would like to express my sincerest gratitude towards my supervisor, Professor Ville Leppänen who gave me the opportunity to conduct my research in his research group. I am grateful for all the supports, encouragements, and inspiration throughout the whole journey. The freedom to choose topic of my interest, the smoothness and flexibility he was offering to the group, made it a joyful place to work for me.

I greatly appreciate the support I received from Associate Professor Seppo Virtanen in the final stage of my PhD with his advice and comments that greatly improved my thesis. I would also like to thank Professor Jouni Isoaho for his support and directing my thesis.

I wish to especially thank Professor Benoit Baudry and Adjunct Professor Martin Gilje Jaatun who kindly reviewed my thesis, and provided fruitful comments and insights. I am as well greatly grateful to Professor Valtteri Niemi, for accepting to act as my opponent.

I would like to warmly express my appreciation towards Professor Asokan for giving me the opportunity of a research visit at the Secure Systems Group in Aalto University. This has been a great chance for my thesis and for me to expand my knowledge, and my professional network. I also would like to thank Dr. Andrew Paverd and Dr. Hans Liljestrand for the very fruitful collaboration we had during this research visit. I also had the chance to meet the amazing people in SSG group and in Aalto University.

I am also very thankful to Professor Pedro Inácio for the research visit opportunity at their research group in Universidade da Beira Interior. This was a great opportunity of collaboration with Professor Inácio and with Bernardo Sequeiros and was a wonderful experience of living in the beautiful city of Covilhã, Portugal.

I would like to thank Professor Mauro Conti, Rehana Yasmin and Reza Memarian for the collaboration on our joint publications.

My colleagues in Software Development Lab, University of Turku, Sampsa

Rauti, Dr. Johannes Holvitie, Jari-Matti Mäkelä, Samuel Laurén, Jukka Ruohonen, Lauri koivunen, and Aki Koivu, very many thanks to all of you for the collaborations, team works, and for making the atmosphere a friendly place to work. I would like to thank Associate Professor Sami Hyrynsalmi, who passed and shared his experiences to me especially when I was in the beginning of the way. His guidance always paved the path for me. Also, my special thanks to Sampsa for reviewing my thesis and very helpful comments, also for all the collaboration we had in the papers we co-authored.

I gratefully acknowledge the funding that I received towards my PhD from Department of Future Technologies, MATTI program, Nokia Foundation, and The Finnish Foundation for Technology Promotion.

I cannot forget to acknowledge my very dear friends who always cheered me up, celebrated each accomplishment with me, and stood by me in the hard days. Thanks to each and every one of you.

My very special thanks go to my beloved family in Iran for standing the distance and for your unconditional love that has always been with me. Thank you for selflessly encouraging me to explore new directions in life, and giving me the opportunities to experiences the experiences that have made me who I am. I also would like to thank my family-in-law here in Finland for their heart-warming kindness that made Finland even a warmer home for me.

And last but not least, my dearest husband, Matti, I feel very grateful for your love and unwavering belief in me. Thanks for supporting me to get through the tough days in the most positive way and always reminding me that "kaikki järjestyy"!

Helsinki, September 2020
Shohreh.

List of original publications

The work discussed in this PhD dissertation is based on the original publications listed below:

Paper I	Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurèn, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen: *Using Diversification and Obfuscation Techniques for Software Security: A Systematic Literature Review.* In Journal of Information and Software Technology (IST). Elsevier, 2018.
Paper II	Shohreh Hosseinzadeh, Samuel Laurén, Sampsa Rauti, Sami Hyrynsalmi, Mauro Conti, Ville Leppänen, *Obfuscation and Diversification for Securing Cloud Computing.* In: Victor Chang, Muthu Ramachandran, Robert J. Walters, Gary Wills (Eds.), Enterprise Security, Lecture Notes in Computer Science 10131, 179–202. Springer, 2017.
Paper III	Shohreh Hosseinzadeh, Samuel Laurén, Ville Leppänen, *Security in Container-based Virtualization through vTPM.* In: 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC 2016), Pages 214-219. IEEE/ACM 2016.
Paper IV	Shohreh Hosseinzadeh, Sampsa Rauti, Sami Hyrynsalmi, Ville Leppänen, *Security in the Internet of Things through Obfuscation and Diversification.* In: International Conference on Computing, Communication and Security (ICCCS), 1–5. IEEE, 2015.
Paper V	Petteri Mäki, Sampsa Rauti, Shohreh Hosseinzadeh, Lauri Koivunen, Ville Leppänen. *Interface diversification in IoT operating systems.* In Proceedings of the 9th International Conference on Utility and Cloud Computing, pp. 304-309. ACM, 2016.

Paper VI	Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, Andrew Paverd, *Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization.* In Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18), pages 42-47. ACM, 2018.

List of publications not included in this dissertation

In addition to the 6 original publications this PhD dissertation is composed of, the author has also contributed to the following publications in the topic area of the dissertation:

- Sampsa Rauti, Samuel Laurén, Shohreh Hosseinzadeh, Jari-Matti Mäkelä, Sami Hyrynsalmi, and Ville Leppänen (2015). *Diversification of System Calls in Linux Binaries.* In: Yung M., Zhu L., Yang Y. (eds) Trusted Systems. INTRUST 2014. Lecture Notes in Computer Science, vol 9473. Springer, Cham.

- Shohreh Hosseinzadeh, Bernardo Sequeiros, Pedro R. M. Inácio, Ville Leppänen, Recent trends in applying TPM to cloud computing. Security and Privacy 3(1), 1–24, 2020.

- Samuel Laurén, Petteri Mäki, Sampsa Rauti, Shohreh Hosseinzadeh, Sami Hyrynsalmi and Ville Leppänen, *Symbol diversification of linux binaries*, World Congress on Internet Security (WorldCIS-2014), London, 2014, pp. 74-79.

- Sampsa Rauti, Lauri Koivunen, Petteri Mäki, Shohreh Hosseinzadeh, Samuel Laurén, Johannes Holvitie, and Ville Leppänen. *Internal interface diversification as a security measure in sensor networks.* Journal of Sensor and Actuator Networks, 7(1), 2018.

- Shohreh Hosseinzadeh, Sami Hyrynsalmi, Mauro Conti and Ville Leppänen, *Security and Privacy in Cloud Computing via Obfuscation and Diversification: A Surve*y, 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), Vancouver, BC, Canada, 2016, pp. 529-535.

- Shohreh Hosseinzadeh, Sami Hyrynsalmi, and Ville Leppänen. *Chapter 14 - Obfuscation and diversification for securing the Internet of*

Things (IoT). In Rajkumar Buyya and Amir Vahid Dastjerdi, editors, Internet of Things, pages 259–274. Morgan Kaufmann, 2016.

- Aki Koivu, Lauri Koivunen, Shohreh Hosseinzadeh, Samuel Laurén, Sami Hyrynsalmi, Sampsa Rauti, and Ville Leppänen. *Software Security Considerations for IoT*, 2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Chengdu, 2016, pp. 392-397.

- Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurèn, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. *A survey on aims and environments of diversification and obfuscation in software security.* In Proceeding of the 17th International Conference on Computer Systems and Technologies 2016, CompSysTech '16, pages 113–120, New York, NY, USA, 2016.

- Sampsa Rauti, Samuel Lauren, Joni Uitto, Shohreh Hosseinzadeh, Jukka Ruohonen, Sami Hyrynsalmi, Ville Leppänen (2016) *A Survey on Internal Interfaces Used by Exploits and Implications on Interface Diversification.* In: Brumley B., Röning J. (eds) Secure IT Systems. NordSec 2016. Lecture Notes in Computer Science, vol 10014. Springer, Cham.

- Rehana Yasmin, Mohammad Reza Memarian, Shohreh Hosseinzadeh, Mauro Conti, Ville Leppänen (2018) *Investigating the Possibility of Data Leakage in Time of Live VM Migration.* In: Dehghantanha A., Conti M., Dargahi T. (eds) Cyber Threat Intelligence. Advances in Information Security, vol 70. Springer, Cham.

- Shohreh Hosseinzadeh, Seppo Virtanen, Natalia Díaz-Rodríguez, and Johan Lilius. 2016. *A semantic security framework and context-aware role-based access control ontology for smart spaces.* In Proceedings of the International Workshop on Semantic Big Data (SBD '16), pages 8:1-8:6, ACM, New York, NY, USA, 2016. ACM.

- Marko Saarela, Shohreh Hosseinzadeh, Sami Hyrynsalmi, and Ville Leppänen. *Measuring software security from the design of software.* In Proceedings of the 18th International Conference on Computer Systems and Technologies, CompSysTech'17, pages 179–186, New York, NY,USA, 2017. ACM.

Abbreviations

AIKs Attestation Identity Keys

API Application Programming Interface

BTB Branch Target Buffer

CoAP Constrained Application Protocol

EK Endorsement key

JSON JavaScript Object Notation

IoT Internet of Things

IPC Inter-Process Communication

JIT Just-in-Time (Compiler)

LBR Last Branch Record

MQTT Message Queuing Telemetry Transport

NIST National Institute of Standards and Technology

HTTP Hypertext Transfer Protocol

IETF Internet Engineering Task Force

OWASP Open Web Application Security Project

PCR Platform Configuration Register

UDP User Datagram Protocol

RESTful Representational State Transfer

ROP Return-Oriented Programming

RQ Research Question

SGX Software Guard Extensions

SLR Systematic Literature Review

TC Trusted Computing

TCG Trusted Computing Group

TCP Trusted Computing Platform

TCP/IP Transmission Control Protocol/Internet Protocol

TEE Trusted Execution Environment

TPM Trusted Platform Module

VM Virtual Machine

VMM Virtual Machine Monitor

WSN Wireless Sensor Network

XML eXtensible Markup Language

6LoWPAN IPv6 over Low Power Wireless Personal Area Networks

Contents

Chapter 1

Introduction

The advancement of cloud computing technology has facilitated the businesses and enterprises with the possibility to deliver services to their customers with lower cost, but higher performance, availability, and scalability. The benefits offered by cloud have made it a highly demanded technology these days. Nonetheless, using cloud services implies that more and more data is stored and processed outside the organization's perimeters, which in fact, raises concerns about the security and privacy of this data. Ergo, this is highly significant that the cloud service providers employ effective security practices to secure their computing infrastructure, to ensure the integrity and confidentiality of the stored/processed data and code, and to preserve the privacy of its users. Some of the taken security measures consider the cloud provider untrusted or malicious from which the users' data need to be protected. In contrast, some other security measures protect the cloud infrastructure from the external intrusions and attacks.

IoT is another wide-spreading technology that is used these days in various public and private sectors for providing services ranging from wearable health monitoring, to smart connected cars and smart grids. IoT encompasses a broad ranges of devices, sensors, actuators, and humans connected to each other through the Internet and exchange data in order to make human lives more intelligent and automated. By the continuous growing trend of this technology, more "things" are connecting to each other every day, collecting and sharing data. It is estimated that the number of connected devices will reach to 75.44 billion by 2025 [6]. However, in spite of the prevalent use of IoT, building security in IoT devices has not been the priority in development and distribution of the IoT devices. According to the report of the review conducted by HP Security Research [7] on the most commonly used IoT devices, 70% of the devices are vulnerable with alarming number of vulnerabilities per device.

The high popularity and wide adaption of these two technologies have in-

creased the significance of security protection of them, as the leakage of data shared and processed by such technologies could have severe consequences. A cloud infrastructure (e.g., storage and computing units) may contain an organization's business critical data or the customers' personal data including social security numbers, credit card information, and addresses. Leakage of such data harms the business and puts user's security and privacy at risk. A breach in an IoT device could open a door for an adversary to enter the network and also could result in the breach of a huge number of similar devices, due to their identical design. IoT devices collect and share a great amount of data, the leakage of which could lead to the loss of user's privacy, security and even safety. Nonetheless, the latest news on the growing number of vulnerabilities, malware, and security attacks on computer systems clearly indicate the current security measures are not sufficient, and there is an urgent need for more robust and sophisticated security approaches.

This dissertation contributes to enhancing the security and trust level in cloud computing and IoT through obfuscation and diversification software security technologies, backed with hardware-based security solutions, trusted computing and Trusted Execution Environment (TEE).

At the application level, we obfuscate and diversify the operating systems and APIs of the IoT devices. Code obfuscation makes the application harder to read and reverse engineer. Software diversification makes the software instances unique which forces the attacker to design individual attack models for each instance. In these ways, conducting a successful attack becomes more challenging, inefficient, and costly from attacker's point of view, and the risk of a massive scale attacks is mitigated. At the network level, we obfuscate and diversify the communication protocols used between the IoT devices, in order to protect the data that is communicated over the network between the IoT devices. In cloud computing we apply obfuscation and diversification at the client-side application.

Nevertheless, we believe that software security measures might not be sufficient in some cases. Therefore, additional hardware security measures make a more robust system to alleviate the risk of both software and hardware attacks. In this regard, in addition to the two discussed software security technologies, we also study hardware-based security solutions, TPM as a trusted computing solution and Intel SGX as a TEE solution. Combination of the software and hardware techniques makes a more complete security approach.

Due to the fact that protecting the cloud software on client-side is not a sufficient solution, we study the use of TPM in cloud computing to leverage hardware-based roots of trust to improve computer security and provide privacy protection and trust. We employ the virtual TPM in a container-based virtualization. Virtual TPM emulates the physical TPM that is integrated on the hardware and presents its functionalities to containers.

We also study using Intel SGX that provides security, trust and isolation for the critical applications and data by creating a secure integrity-protected processing area that guarantees the confidentiality and integrity of the code and data that is loaded in it. However, since SGX is prone to some side channel-attacks, we employ obfuscation as a software security technique. We apply control-flow obfuscation on the enclave programs of the SGX architecture, and mitigate the branch-shadowing side-channel attack on SGX.

The dissertation is organized in five different chapters:

- Chapter 1 presents an introductory note on this PhD dissertation and organization.
- Chapter 2 presents the preliminaries and the background on the concepts discussed in this dissertation. This chapter first focuses on presenting the background on the studied execution environments in this dissertation (cloud computing, IoT) and general status of security in these environments. The chapter then continues with introducing the software and hardware security techniques used in the dissertation (obfuscation, diversification, trusted computing, and TEE) to enhance the security and trust level in cloud computing and IoT.
- Chapter 3 describes in detail the contributions that this dissertation has made in enhancing the security and trust for cloud computing and IoT, regarding the background presented in Chapter 2. This chapter, starts with introducing the objectives and motivation of this dissertation, the research questions, the research methodology, and it continues by presenting how the dissertation answers the research questions through the original publications included in this dissertation.
- Chapter 4 presents the concluding remarks, contributions made by this PhD dissertation, and the future research direction.
- Chapter 5 presents the original publications included in this PhD dissertation, Publication I - Publication VI.

Chapter 2

Preliminaries

This chapter presents background preliminaries of the terms, concepts and technologies discussed in this dissertation, including cloud computing, IoT and the general status of security in these environments. The chapter continues with presenting the software security solutions (obfuscation and diversification), and the trust assurance solutions (trusted computing and TEE) that are used to improve security and trust in the stated environments, in the original publications and the subsequent chapters of the dissertation.

2.1 Cloud Computing

Cloud computing has become a popular technology through which organizations and service providers are delivering services to their customers in a cost-effective and convenient way. This section presents the background on cloud computing: the definition, architecture, deployment models and the discussion on the current security status of cloud computing.

2.1.1 Definition

Cloud computing is the delivery of computing services such as storage, servers, databases, software, and networking over the Internet, on a pay-per-use basis. The following is the definition provided by the National Institute of Standards and Technology (NIST) for cloud computing [47]:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models."

In the stated definition five essential characteristics are mentioned for cloud computing. These characteristics are as follows [47]:

1. *On-demand self-service*: computing capabilities could be provisioned unilaterally by the consumer when required.

2. *Broad network access*: computing capabilities are widely available over the network using thick and thin clients. Clients refer to the applications that run on workstations or computers and rely on a server to carry out operations. A thick client is a full-featured computer that often provides rich functionality and does not require continuous communication with the central server. In contrast, a thin client is designed small in size and the main data processing occurs on the server.

3. *Resource pooling*: using a multi-tenant paradigm, the computing resources are pooled to serve a multitude of consumers. In this paradigm, there exists a location independence concept which means that service consumer may not necessarily have knowledge or control over the physical location of the resources (such as processing, storage, and memory).

4. *Rapid elasticity* refers to scalable provisioning and the ability of providing scalable services. Rapid elasticity provides users with additional space and resources on the cloud upon their request.

5. *Measured service*: the cloud service provider monitors, controls, reports and optimizes the offered resources in a transparent manner to both provider and consumer of the services.

2.1.2 Deployment Models

Cloud computing is an evolving technology with new capabilities and services that have remarkable benefits compared to traditional service providing approaches. The services are delivered with lower cost (in usage as it is pay-for-use, in disaster recovery, in data storage solutions), greater ease, less complexity, higher availability and scalability, and faster deployment. These compelling benefits have motivated the enterprises to adopt cloud solutions in their architectures and deliver their services over cloud. Depending on the need of the enterprise and how large the organization is, four different deployment models are available, including public, private, community, and hybrid clouds [1, 45, 47]. In a *public (or external) cloud*, a third party vendor is responsible for hosting, operating, and managing the cloud. A common infrastructure is used to serve multiple customers, which means that the customers are not required to acquire any software, hardware, and network devices. This makes the public cloud a suitable model for the enterprises that wish to invest less and manage the costs efficiently. The security in a

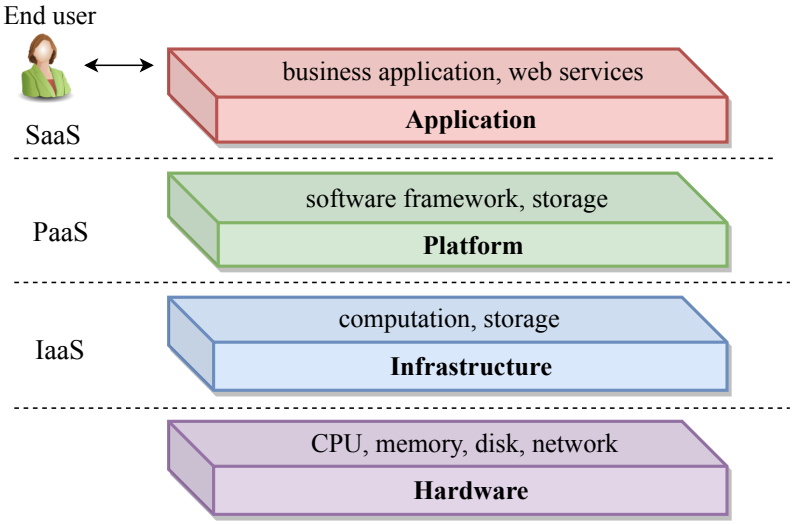


Figure 2.1: Cloud architecture and resource management at each layer

public cloud is managed by the third party, which leaves less control for the organization and its users over the security. The contrary solution is *private (or internal) cloud*, where the organization's customers are in charge of managing the cloud. The storage, computing, and network are dedicated to the customer owning the cloud, and not shared with other customers. This enables the customers to have a higher control on security management and have more insight about logical and physical aspects of the cloud infrastructure. *Community cloud* refers to the type of clouds that are used exclusively by a community of customers from enterprises with common requirements and concerns (e.g., policies, security requirements, and compliance considerations). The last model is *hybrid cloud* that is the composition of several clouds (private, public, and community). According to the needs and budget of the enterprise and how critical its resources are, a suitable deployment model is chosen that can serve the enterprise's needs the best way [45].

2.1.3 Cloud Architecture and Service Models

The cloud computing services are offered to the customers in three different models [23, 47], and depending on the need of the consumer/enterprise, a suitable delivery model is deployed and adapted: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Figure 2.1 illustrates the cloud architecture and the resource management at each layer [23]. The *IaaS* model offers storage, network and fundamental computing resources to the consumers, so that they could deploy and run software such as operating systems and applications. The consumers do not

have control over the underlying infrastructure, however, they could manage and control the storage, operating system and deployed applications, and some of the networking components. The *PaaS* model offers an integrated environment to the consumer for deploying, building, and testing applications using programming languages, tools and libraries supported by the service provider. The consumer has no control over the underlying cloud infrastructure (e.g., server, operating system, storage). In the *SaaS* model, the main services offered by the service providers to the consumer are the applications that are hosted and executed on the cloud and are available to the customers through the network, typically over the Internet. In this model, the consumer has no control over the underlying infrastructure including servers, operating system, network, and storage.

2.1.4 Virtualization Models

During the past decade various virtualization mechanisms have been developed, including hypervisor-based virtualization and container-based virtualization. In hypervisor-based virtualization, a hypervisor (Virtual Machine Monitor (VMM)) is running on top of the server hardware enabling multiple VMs to share virtualized hardware resources. This, in fact, means that different operating systems (e.g., Linux, Windows) can run on top of the hypervisor on the same physical platform. However, in container-based virtualization (also referred to as operating-system-level virtualization), virtualization happens at the operating system level rather than the hardware. That is to say, only one operating system can run on top of the platform, and the containers share the kernel of this operating system. In other words, operating system virtualization allows running a multitude of execution environment instances on a single kernel. Figure 2.2 illustrates the architecture of these two virtualization models. A container is considered as a lightweight operating system that runs inside the host system, with the instructions that are native to the core CPU. This, in effect, eliminates the need for just-in-time compilation or instruction level emulation [25].

Containers and VMs are similar in the sense that they offer resource isolation and allocation benefits and meanwhile, different in function. In the container-based virtualization the operating system is virtualized, while in the hypervisor-based virtualization the hardware is virtualized. There are advantages and disadvantages associated with each of these virtualization models. Due to the fact that containers are more lightweight in comparison to VMs, they have lower performance overhead, take less storage, and they boot faster than the guest operating system. On the other hand, the container-based virtualization has lower flexibility considering that the base operating system is shared and the containers are hosted with the same operating system that the hosting platform uses.

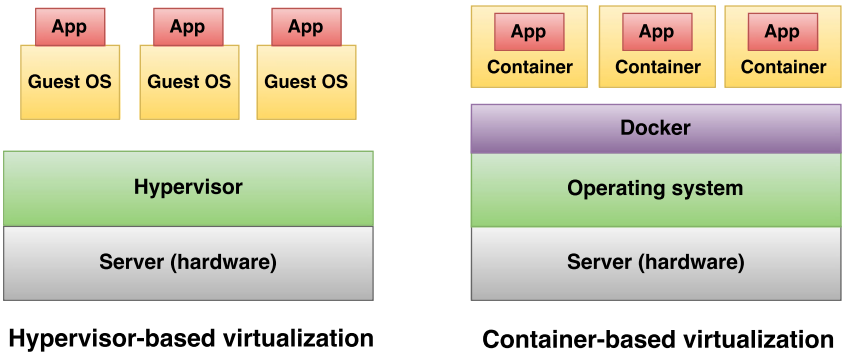


Figure 2.2: Hypervisor-based vs. container-based virtualization

2.1.5 Status of Security in Cloud Computing

The convenience and efficiency offered by cloud computing have encouraged more users to turn towards this technology. Cloud service providers have a strong incentive of keeping their software proprietary due to competitive advantages, and as well are reluctant to report bugs and security incidents to maintain the reputation of their business. In addition, in cloud computing where the computing (storage and services) happens outside the organization's premises, the customers are neither aware of the location where their data is stored/processed, nor have any knowledge of the way cloud infrastructure is being managed. The client organizations with more strict data protection policies are unwilling to confide their data to service provider, with the concern that their data might be accessed by an unauthorized party (including cloud provider itself), and might be utilized in an unwanted manner. Furthermore, due to the nature of virtual environment, in order to support the cloud's multi-tenancy attribute, some additional layers are added to the architecture: the hypervisor that resides between the hardware and the operating system, and the operating system which is running on the VMs on top of the hypervisor. In addition to the risk of security attacks that existed in the traditional computing systems, the added layers have introduced new attacks and expanded the attack surface. These concerns on the security, confidentiality and integrity of the data have become the primary issue of trust from clients towards the cloud provider, which therefore has slowed down the pace of cloud computing adoption into the businesses.

The role of security community is to provide strong security measures to firstly secure cloud infrastructure in its all layers so to protect the data being stored or processed on cloud provider side from external threats. Secondly, they should provide and develop security measures to protect the customers' data from cloud service provider, so to guarantee the confidentiality and integrity of data even if the cloud provider is compromised or malicious.

2.2 Internet of Things

2.2.1 Definition

Decades after the advent of the Internet, the emerging era of ubiquitous computing began and the continuous connectivity is no dream anymore. One of the prominent elements of pervasive computing is Internet of Things (IoT) or Internet of Everything. The term was used for the first time by MIT Auto-ID Labs in 1999 [17]. IoT embeds computational capabilities into the everyday used objects, and enable their connectivity to each other and to the Internet from anywhere at any time. The term Internet of Things is composed of two significant pillars of "Internet" and "Things". The notion "Things" encompasses any object that is able to connect to the Internet, including sensors, smart devices, and any other element that is capable of communicating with other entities. IoT is regarded as the third revolution in information technology after Internet and mobile communication networks. Today, IoT is used in multitude of public and private sectors in various applications including healthcare, smart spaces, wearable data monitoring systems, smart lighting, and farming.

2.2.2 Operating System and Software on IoT Devices

IoT comprises a wide variety of heterogeneous components with varied functionalities and computing capabilities ranging from sensors with lightweight 8-bit micro-controllers, to powerful 32-bit processors (e.g., PCs and smartphones). On that account, the designed operating system and the software should be adaptable by all ranges of objects participating in the network including the low-powered ones. Moreover, in addition to supporting the functionality of the ubiquitous devices, the software should also be compatible with the limitation of these devices, in terms of memory, computational power, and energy capacity. Baccelli et al. [14] discuss that the software designed for IoT devices should be a) designed considering the heterogeneous hardware constraints: the IoT software should operate with low complexity and have fairly low memory requirement; b) programmable: the IoT software should provide a standard application-program interface (API) and support the standard programming languages (such as C, C++) for application developers; and c) autonomous: for energy efficiency purposes, the software should consider sleeping cycles for energy saving at the times that the hardware is in idle mode. Moreover, due to the fact that IoT is deployed in critical systems, it is essential that the operating system functions reliably. These requirements motivated the developers to build operating systems and software that are compatible with all variety of IoT devices with diverse capabilities and capacities, that could both leverage the capabilities of the more powerful devices and could also be run on the power-

restricted devices. Table 2.1 lists some of the operating systems designed for such low-end embedded devices. The most widely used operating systems for IoT devices are Contiki [26] and TinyOS [41]. Contiki [26] is an open-source operating system, developed in C language to operate on low-power memory-restricted devices (by requiring only a few kilobytes of memory). Through supporting networking standards (e.g., CoAP, IPV4, and IPV6) Contiki enables the low-power microcontrollers to connect to the Internet. TinyOS [41] is as well an open-source operating system that is developed in nesC [9] language (an extension of C). Similar to Contiki, TinyOS is multithreading, event-driven, and is designed according to a component-based programming model with monolithic structure. TinyOS is specifically designed for the sensor network devices with constraint resources (e.g., 512 bytes of RAM and 8Kb of memory).

Table 2.1: Operating systems for embedded devices

OS	**Overview**	**Characteristic**	**Lang.**	**Open source**
Contiki	designed for WSN and memory-efficient embedded systems network	Modular structure, multi-tasking, multi-threading, event-driven	C	✓
TinyOS	Intended for the low-power wireless devices	Monolithic structure, multi-threading, event-driven	NesC	✓
RIOT OS	Real-time	Modular structure, multi-threading	C, C++	✓
Mantis	Designed for WSN. It presents C API with Linux and Windows development environments.	Threads	C	✓
Nano-RK	Equipped with a light-weight resource kernel and networking support for WSN.	Threads	C	✗
LiteOS	UNIX-like OS for WSN	Threads and Events	LiteC++	✓
Free RTOS	Real-time OS designed for embedded devices	Multi-threading	C	✓
Linux	Various Linux distributions can run on IoT devices, e.g. Raspbian and Ubuntu Core	Monolithic structure, event-driven	C, C++	✓

2.2.3 Protocols Used for Connection of IoT Devices

Considering TCP/IP as the de facto standard for the communication networks, some are on the opinion that it could also be used in IoT in the future. However, at this point the low capacity of the IoT devices makes the deployment of IPv6 for connecting such devices challenging. Therefore, the Internet Engineering Task Force (IETF) has presented protocols that are adaptable to this environment, with the assistance of which the normal IP-based devices (e.g., PCs and smartphones) can connect to the low-end devices. Since the IoT devices may use different protocols to communicate, the protocols should be translated to a standard protocol using XML, JSON, or RESTful APIs. The following ones are some of the most commonly used protocols for connecting the IoT devices to each other through the Internet:

- User Datagram Protocol (UDP) [49] is the most commonly used protocol at the transport layer for the low power networks.
- Constrained Application Protocol (CoAP) [53] is an application-layer protocol built on UDP and is compatible with resource constrained nodes, and replaces HTTP as it has higher overheard for IoT.
- IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN) [8] is a protocol that comes as an adaptation layer between the link and network layers, and allows transmission of packets over IEEE 802.15.4-based network. It presents packet fragmentation and header compression to decrease datagram size.
- Message Queuing Telemetry Transport (MQTT) [34] is a publish/-subscribe messaging protocol for communication of the devices and servers. It is a many-to-many protocol, i.e., the messages are transferred between clients via a central broker. The protocol is designed on top of TCP/IP, thus the connection can be encrypted with SSL/TLS.
- Advanced Message Queuing Protocol (AMQP) [58] is an application-layer protocol that reuses the underlying transport models (e.g., TCP/IP and UDP) and connects servers together. AMQP offers flexible routing and queuing system.
- Extensible Messaging and Presence Protocol (XMPP) [2] is protocol based on XML that provides instant messaging and presence functionality services. XMPP is used to connect devices to servers.

2.2.4 Security Status of IoT

From the advent of IoT technology, the focus of the service providers principally has been on operability and availability of the IoT services. Often,

security of the devices and privacy of their users were afterthoughts. This is while IoT has gained more and more popularity and more sensors, actuators, and devices are connecting to each other and to the Internet. The number of connected devices is projected to reach 75.44 billion by 2025, which is about 5 times increase in 10 years time (2015 - 2025) [6]. These objects/devices/sensors collect a great deal of data and share them over the Internet. Some of these data contains sensitive information about users such as personal, financial, and the physical location of the users. Leaking of such data could not only result in violation of the users' privacy, but could also endanger their safety and security. HP Security Research conducted a review on the most commonly used IoT devices, and the report shows that 70% of the devices are vulnerable with very high rate of vulnerabilities per device [7]. These vulnerabilities include Heartbleed, denial of service, weak passwords, and cross-site scripting. Considering the fact that IoT is established on the Internet, it is prone to the traditional security risks for the Internet. In addition to those risks, the dynamic nature of the IoT environment, along with the heterogeneity and large scale of devices, make the traditional security issues more critical and also present new security challenges. OWASP [11] has listed the following as the 10 top vulnerabilities that IoT devices suffer from: easily guessable and brute-forced passwords, insecure network services, insecure ecosystem interfaces, lack of secure update mechanism, using outdated and insecure components/libraries on the device, lack of sufficient privacy protection mechanism, insecure transmission and storage of data (no encryption and access control mechanism), absence of device management, insecure default setting, and absence of physical hardening measures.

Securing the IoT devices and protecting the collected and shared data is exceedingly significant. However, special characteristics of IoT devices have made securing of them a challenging task compared to the traditional computing devices. These special characteristics include low capacity of devices in terms of memory and computational power, heterogeneity of devices with different capabilities, scalability of network, wireless connection of devices to network (via Wi-Fi, ZigBee, Bluetooth), mobility of the devices in the network, and the single-purpose design of the embedded IoT devices. Therefore, it is essential that the security measure that is designed to protect the security and privacy in IoT is lightweight, tolerable by the devices, and is compatible with the limitations of the IoT devices.

2.3 Obfuscation

Code obfuscation is the process of scrambling and transforming the code to an unintelligible form that is more complex, and harder to read and comprehend. The obfuscated code is syntactically different, but still functional

and semantically equivalent to the original code [20]. With the help of obfuscation transformations, even if an adversary obtains the program's source code, it takes more time and energy to comprehend the code and reverse engineer it. Obfuscation does not guarantee that the program will not be tampered with or reversed engineered, but it advances the level of defense by increasing the effort and cost for the attacker to learn the functionality of the obfuscated program.

Figure 2.3a shows a simple piece of C code and Figure 2.3b shows the obfuscated version of this code using a free online obfuscation tool [10]. The obfuscated version of the code is not easy to read and understand. Certainly, with given resources and time the attacker may be able to comprehend and reverse engineer the code, albeit requiring higher time and energy.

(a)

```
#include <stdio.h>
int main()
{
   int i, n, a = 0, b = 1, nextTerm;
   printf("Enter the number of terms: ");
   scanf("%d", &n);
   printf("Fibonacci Series: ");

   for (i = 1; i <= n; ++i)
   {
      printf("%d, ", a);
      nextTerm = a + b;
      a = b;
      b = nextTerm;
   }
   return 0;
}
```

Obfuscation

(b)

```
#include <stdio.h>
int main(){int
o_29fb4af0a1adf7ee01a3b3dd06bbe963,o_c3fc214552f8031f1447db420dff1c96,o_eab5836874947171c8ad490372fe6348=
(0x0000000000000000 + 0x0000000000000200 + 0x0000000000000800 -
0x0000000000000A00),o_31b27256bfe8509585c3676eed39aaca=(0x0000000000000002 + 0x0000000000000201 +
0x0000000000000801 - 0x0000000000000A03),o_ae6b7175f17dcdedea722a581a2ca6b9;printf("\x45""n\164e\x72"" \164h\x65""
\156u\x6D""b\145r\x20""o\146
\x74""e\162m\x73"":\040");scanf("\x25""d",&o_c3fc214552f8031f1447db420dff1c96);printf("\x46""i\142o\x6E""a\143c\x69""
\123e\x72""i\145s\x3A"" ");for (o_29fb4af0a1adf7ee01a3b3dd06bbe963 = (0x0000000000000002 + 0x0000000000000201 +
0x0000000000000801 - 0x0000000000000A03);(o_29fb4af0a1adf7ee01a3b3dd06bbe963 <=
o_c3fc214552f8031f1447db420dff1c96) & !!(o_29fb4af0a1adf7ee01a3b3dd06bbe963 <=
o_c3fc214552f8031f1447db420dff1c96);++o_29fb4af0a1adf7ee01a3b3dd06bbe963){printf("\x25""d\054
",o_eab5836874947171c8ad490372fe6348);o_ae6b7175f17dcdedea722a581a2ca6b9 = o_eab5836874947171c8ad490372fe6348 +
o_31b27256bfe8509585c3676eed39aaca;o_eab5836874947171c8ad490372fe6348 =
o_31b27256bfe8509585c3676eed39aaca;o_31b27256bfe8509585c3676eed39aaca =
o_ae6b7175f17dcdedea722a581a2ca6b9;};return (0x0000000000000000 + 0x0000000000000200 + 0x0000000000000800 -
0x0000000000000A00);};
```

Figure 2.3: Excerpt from obfuscated piece of C code: a) original code, b) obfuscated version of the same code

In the literature, there has been a large body of research on the various obfuscation mechanisms [21], each of which has different target for applying obfuscation transformation, and at various phases of the software development life-cycle [33]. According to the taxonomy presented by Collberg et

al. [21] the target of obfuscation could be control flow, data, and program layout. For achieving each of these obfuscation transformations, there exist a wide range of mechanisms. *Control flow obfuscation* is one very common way of concealing the program flow. To obfuscate the program's control flow, opaque predicates could be used [22]. An opaque predicate is a Boolean expression in a program code that is designed to be executed always the same way. Although the outcome of this execution is always known to the obfuscator in a priori, it still is evaluated at execution time to make the code analysis harder and more cumbersome. Bogus (dead/gray/dummy code) insertion [27, 59] is another common control flow obfuscation technique that is used to confuse the code analyzing tools, by adding a piece of code that is never executed. *Data obfuscation* attempts at obscuring data and concealing data structure of a program. Some of the obfuscation mechanisms that fall in the category of data obfuscation are class transformation [56], and array transformation [24]. The third category is *layout transformation* which is a type of obfuscation that targets the program's layout structure, for instance through renaming the identifiers [35], Instruction Set Randomization (ISR) [57], and Address Space Layout Randomization (ASLR) [44].

Code obfuscation has been proposed and used by software developers for various purposes [33], including protecting secrets and data in a program, protecting protocols from spoofing, protecting program from tampering and illegitimate modification, making code analysis difficult, protecting digital watermarks and birthmarks in programs, and protecting mobile agents. Like many other security measures, obfuscation has also been used by malware developers to conceal the malware's malicious code from being detected by the scanners [64].

2.4 Diversification

Software diversification is changing the structure and *internal interfaces* of a program in order to generate unique diversified versions of this program. Unique instances of the software are distributed among the users. These instances are all functionally the same, but differently diversified. This means that diversification breaks the monoculture software distribution model and introduces multiculturalism in software deployment process. The security benefit of the multicultural software deployment model is impeding the risk of large-scale attacks. Even supposing that the attacker manages to crack one instance of the software and utilize some non-public internal interface, on account of the fact that software instances are unique, the attacker cannot access other instance. In other words, the attack model must be designed individually for each particular software instance and the same attack model does not work on all software instances. Therefore, diversification is a very

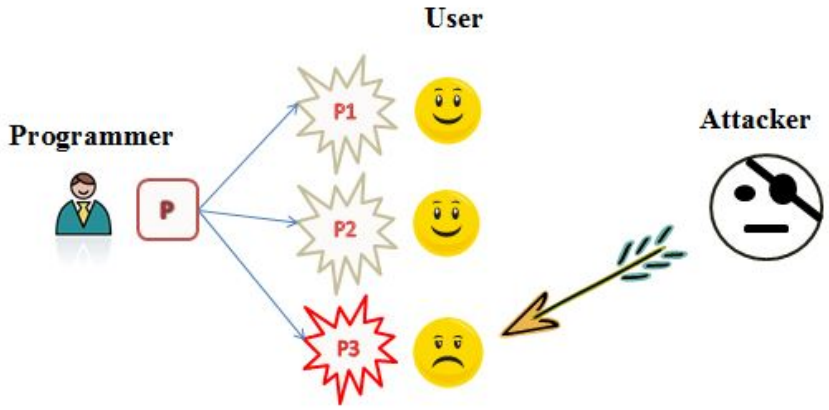


Figure 2.4: Software diversification

effective technique for securing largely distributed systems and also the systems in the current era of operating system and interface monoculture. Figure 2.4 shows the unique design and distribution of the diversified software instances, and how diversification thwarts the risk of large-scale attack.

In order to access the resources of a system, interfaces are used. Diversification alters the internal interface(s) to secret ones. Diversifying the internal interfaces of the software creates a unique secret that is propagated only between the legal "clients" of the interface (i.e., they are also diversified to be compatible with inner layers), to maintain their communication between each other. The diversification secret is kept private from illegitimate parties to prevent their access to the diversified resources.

Interface diversification introduces proactive protection against malware, especially when it is applied throughout all the interface layers. Malware (malicious software) is a piece of software that is designed to run its code on a user's computer to disrupt its operation or manipulate the system in the way that the attacker desires [55]. To do this, malware needs to gain knowledge on the system, its environment, and how to interact with it and access its resources. Software diversification modifies the internal interfaces of the software and makes it challenging for the malware to get such knowledge. Consequently, the malware code becomes incompatible with the system's environment and when not able to interact with it, it becomes ineffective. To achieve this, many different diversification techniques have been proposed that each target various parts in software layer. Depending on the need of the system, different diversification mechanisms could be employed. Diversification mechanisms could be as simple as name changing or a more complex technique such as system call diversification.

System calls are the programmatic ways for a program to request services

from the operating system kernel. System calls as part of overflow attacks could be misused by a malware to access the operating system and carry out its malicious intend. Diversification of the system calls could conceal their interface and make them inaccessible by the malware. Symbol diversification [39] diversifies the indirect library entry points to the system calls by renaming the symbols in Linux binaries. This makes it difficult for the malware to use these entry points and system's resources, because it does not know the diversification secret (the function that relates the original and new symbol names). Liang et al. [42] propose a system call randomization approach to counter code injection attacks. This type of attacks uses system calls to inject code. By randomizing these system calls, the attacker who does not know the randomization algorithm key, will inject a code that is not randomized in the same way that the target process has been randomized, so it is not valid for the de-randomized module. Thus, the injected code fails to execute without correctly calling the system calls. Address Space Layout Randomization (ASLR) [44] is another practical and effective diversification technique that protects the operating system from buffer overflow by randomizing the location of the executables in the memory [35]. This, randomizes and reshuffles the address space locations of a process (the base of the executable, heap, stack, and library position). Here, the memory could be seen as an interface that after diversifying it, the attacks that rely on the known layout structure are prevented.

Software diversification has many parallels with code obfuscation. Whilst the main goal of these two software security approaches are different, many of the techniques developed for software obfuscation can be parameterized for software diversification purposes. This means that obfuscation could be applied to generate diversified versions of the code, and obfuscation could be applied to make diversified software less understandable and distinguished from equivalent versions. These two techniques do not aim to eliminate software vulnerabilities, but make it difficult for an attacker to exploit these vulnerabilities and conduct a successful attack. In the literature, there exist a wide range of studies on applying these two techniques for securing various execution environments [32] such as cloud computing [37, 50], IoT [29, 30], mobile and embedded systems. The studies show that the two techniques have been successful in mitigating the risk of a wide range of attacks [32] including buffer overflow, injection attacks, Return-Oriented Programming (ROP) attacks, and JIT spraying attacks. Diversification and obfuscation could be applied on different parts of software at different phases of development process: design and implementation, compilation and linking, loading, installation and update, and execution. The most common phase is compile time (i.e., at the time the program is compiled and linked) [33].

It is worthwhile discussing here how obfuscation and diversification differ from security through obscurity. Security through obscurity protects the

code/secret/asset by concealing it. This means that the asset is safe until it is not discovered. When it is discovered, it is no longer safe, and the system is vulnerable. This is while that obfuscation does not try to hide the program but instead secures the program, even in the case that the source code or the binary code of the program is revealed. This means that even if the attacker gets access to the code, he/she cannot read/understand it, because it is obfuscated in a way that the attacker does not know. For instance, in source code obfuscation the code is not hidden but it is obfuscated in a way that it is difficult to comprehend and reverse engineer.

Diversification removes the knowledge necessary to interact with the environment and access the resources, i.e., the previously publicly known interfaces towards the system resources are removed, and thus, the system is hardened. For instance, in symbol diversification [39] the indirect library entry points to the system calls are renamed. This means that the old names are not hidden, but they do not exist anymore.

Therefore, security through obscurity could be valuable to add a valid layer of security, but it cannot be a sufficient and strong security mechanisms by itself and cannot substitute the security for secrecy.

2.5 Trusted Computing

Trusted computing (TC) is a technology that leverages a hardware-based roots of trust to improve computer security through hardware enhancements and alteration of the associated SW. TC creates a secure environment referred to as Trusted Computing Base (TCB) which provides privacy protection and trust. TCB assures the security of the system and guarantees that it is behaving in the expected manner. Different hardware manufacturers have promoted and developed specifications to protect computer resources from the access of malicious entities. Trusted Platform Module (TPM) [13] is a tamper resilient co-processor chip that is designed by Trusted Computing Group consortium [12], to provide security solutions to its hosting platform. These solutions include trusted boot, secure storage, integrity measurement, remote attestation, and cryptographic functionalities. Today, a large number of computing devices are equipped and shipped with TPM and benefit from the security guarantees that it offers. TPM specification version 1.2 was first published in 2011, and a newer version of it, TPM 2.0, was released to the market with stronger cryptographic capabilities and algorithm supports. TPM provides Root of Trust for Storage (TRS), Root of Trust for Reporting (RTR), and Root of Trust for Measurement (RTM) [62]. RTS offers a protected and secure repository to store the sensitive data and cryptographic keys. It also contains a cryptographic engine to securely conduct lightweight cryptographic operations. RTR offers a pro-

tected environment and interface that is responsible for managing identities and signing the assertions. It cryptographically binds the information to a specific device, in order to prove the integrity and non-repudiability of the information. RTM provides the measurements on the platform's trust status. These services offered by TPM, make the identification, authentication, integrity verification, and encryption of a device feasible.

Remote attestation which is the primary attribute provided by TC technology extends the trust from the lower levels, towards the upper levels and applications. This step-by-step verification of trust is known as the chain of trust, in which all the component are considered untrusted and are measured before they are loaded or executed. A TCB measures the integrity of the platform both at the boot time and at the run time. At the time of system boot, it needs to be assured that the system is being booted correctly, and it is running the appropriate operating system. This is achieved by creating a chain of trust starting from the Core Root of Trust for Measurement (CRTM). CTRM is a trusted code in the BIOS boot block that measures the integrity values of other components, and stays unaltered during the whole platform's lifetime. CRTM is an extension of normal BIOS that is run to measure the integrity of other parts of the BIOS block. The BIOS measures the hardware and the bootloader, and then it passes the control to the bootloader. The bootloader measures the kernel image of the operating system and passes the control to the operating system. Based on the taken measurement values in each of the steps in the boot process, the appropriate Platform Configuration Register (PCR) value is updated [63]. Storing a new value to the PCR works by extending the existing value with a new value. TPM PCRs are the protected memory locations for storing sensitive information, such as integrity measurement values that attest the system's integrity. Besides the strong isolated storage that the TPM offers, it also holds a set of keys that assist at the time of attestation, Endorsement key (EK) and Attestation Identity Key (AIK). EK is an RSA key pair that is used for cryptography operations such as digital signature operation. This key is a pair of public and private key that is generated at time that TPM is being manufactured. The private part of the key is tied to TPM and never leaves it. EK is used for signing the information that is generated inside the TPM, such as PCR values. AIKs are the keys that are used at the time of remote attestation, to preserve the privacy of the platform's identity. When a platform (attester) receives remote attestation request from a verifier, it sends out an integrity report that is composed of PCR values and their digital signature computed with an AIK. Due to the fact that the private part of the AIK's key pair never was released from TPM, a certified authority can certify the authenticity of it. This is a guarantee for the integrity and authenticity of the report [62].

In recent years, hardware virtualization technology has emerged rapidly

and has reduced the costs associated with the ownership of computer systems. This technology has especially been advantageous for systems that share hardware platform among multitude of software workloads with the aim of cost reduction and utilization improvement. Cloud computing is one of these systems that have benefited from hardware/platform virtualization. The added benefits have also introduced security concerns due to the shared resources. The software workloads that are sharing the same hardware should be kept separate from each other to protect their security and maintain the software integrity. Virtual Machine Monitor (=hypervisor) is a proper solution for isolating these workloads, and TC is a suitable technology to provide hardware based root of trust, guaranteeing software integrity and mitigating the software attacks. Hence, combination of these technologies is a well-fitting solution in this scenario. Virtualization of the TPM was proposed [16] to make TPM functionalities available to all VMs running on a platform, in a way that there is only one physical TPM per platform which is virtualized. Each VM receives its own virtual TPM (vTPM) instance and these vTPM instances act just as physical TPMs and offer the same functionalities as the hardware TPM. Due to the fact that vTPM is software based, it has more flexibility comparing with physical TPM.

2.6 Trusted Execution Environment

A Trusted Execution Environment (TEE) [51] is a secure integrity-protected processing area inside the main processor with memory and storage capabilities. It runs in an isolated environment in parallel to the operating system, and guarantees the confidentiality and integrity of the code and data that is loaded in the TEE. The trusted applications that run in TEE have access to the processor and memory of the device. Hardware isolation secures these applications from the other applications installed by the user that run in the operating system. Cryptographic software isolation inside these trusted environments protects different applications from each other. The primary concepts of a TEE are security, trust, and isolation of critical data. TEE enables device identification through root of trust which enables the stakeholders to identify the authenticity of the device that they are interacting with. Also, it cryptographically protects the data that is processed and the applications that are executed in it.

Intel® SGX [4, 5] is a hardware-based TEE solution developed to provide isolated execution. SGX is a set of new instructions and modifications to the memory access, added to the Intel® architecture that enable an application to instantiate a highly protected container called enclave. Enclave is referred to the protected area in the address space of an application that guarantees integrity and confidentiality, even in the presence of malware. This protected

memory area is accessible only by the software that resides in the enclave, and access is prevented for the applications outside it, even for the privileged software such as hypervisor, BIOS, or operating system [46]. The memory area utilized by an enclave is encrypted in order to protect application's secrets from adversaries with access to the memory. Some instances of the typical use cases of enclave are password managers, password input, and cryptographic operations. Figure 2.5 shows the enclave secure container within the application's virtual address space [46].

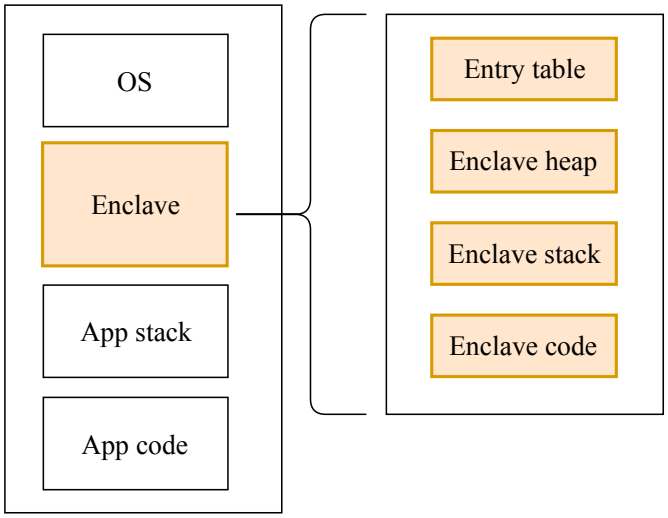


Figure 2.5: Excerpt of an enclave within application's virtual address space

In the recent years, Intel SGX has received considerable attention by virtue of the fact that it could be a practical solution for various domains including trustworthy public cloud. Confidentiality and integrity assurance have always been a major concern and obstacle on cloud adoption. These concerns could be alleviated through the use of Intel SGX.

Despite of the great advantages that Intel SGX has introduced, it has recently shown to be prone to several side-channel attacks that attempt to extract some information about the program running in an enclave. One of these side-channel attacks is the branch shadowing attack [40] that aims at learning the secret-dependent control flow of the code running inside an enclave. To do this, the attack misuses the Branch Prediction Unit (BPU) of the CPU to identify whether an instruction is executed/taken or not, and also the target of the branch. BPU is a feature in modern processes to improve performance through instructions pipelining before exact branching decisions are known. In other words, based on the recent branch history, BPU speculates whether or not a branch is taken, and also the targets of indirect branches. Similar to branch-shadowing attack that uses BPU

and CPU's speculative execution feature, Meltdown [43] and Spectre [36] also misuse the critical vulnerabilities of modern CPUs to get hold of the secrets that are stored in the memory of other running applications, such as passwords stored in the browser or password manager, personal emails, photos and business-critical documents stored on the victim's machine.

Page fault (=control-based) side channel attacks [61] are critical side-channel attacks that allow a malicious operating system to take control over the SGX program's execution and to extract large amounts of sensitive information from the protected application. To conduct this type of attack, the malicious operating system interrupts the execution of the enclave, unmaps the target memory pages, and resumes the execution of the enclave. To unmap the memory, the operating system induces page-fault traps through restricting the access to particular data pages or code. At the time when a data object on that particular data page is accessed or a function on that particular code page is called, a page-fault is triggered and the operating system is notified. By observing the pattern and sequence of accessed addresses of data and control transfers, the malicious operating system can infer the sensitive data.

A cache-based side channel attack [19] extracts the secret of an enclave program (e.g., RSA key) through monitoring the access patterns of the caches that are shared between enclave and untrusted software. This monitoring could be done through Prime+Probe cache monitoring technique [48]. In this technique, the adversary fills up all the cache lines. Then the victim accesses a cache line. The adversary detects whether or not the cache line is used by the victim through checking if its cache entry has been evicted.

Intel has stated that these side-channel attacks are beyond the scope of SGX and it is the responsibility of the developer to mitigate the risk of such attacks [3]. In this regard, researchers and developers have developed several defense mechanisms to mitigate these side-channel attacks, such as T-SGX [54] and SGX-Shield [52] as mitigating solutions to the controlled-channel attack, DR.SGX [18] to mitigate cache attacks on enclave, and mitigating solutions to thwart the branch-shadowing attacks [31].

Chapter 3

Contributions of the dissertation

This chapter is organized in five different sections and describes in detail the contributions this dissertation makes in providing security and trust for cloud computing and IoT. Section 3.1 presents objectives and motivations of this dissertation (M1 - M5) to conduct further research and tackle the security issues. Section 3.2 introduces research questions (RQ1 - RQ4) devised in order to follow the motivations and achieve the objectives of dissertation. Next, in Section 3.3 we present the research methodology chosen for this dissertation, and in Sections 3.4 we explain the process of answering research questions through the original publications included in this dissertation. Finally, Section 3.5, presents a detailed description of the contributions that each individual publication has made to this dissertation.

3.1 Motivation and Objective of the Research

The following, M1 - M5, are the motivations of conducting this research dissertation in the studied domain, security and trust in cloud computing and IoT. M1 - M3 are the motivations on the choice of execution environment, and M4 - M5 are the motivations for the choice of security approach:

- *M1: Improving security in cloud computing*
- *M2: Improving trust in cloud computing*
- *M3: Improving security in IoT*
- *M4: Improving the security of software through obfuscation and diversification techniques*
- *M5: Improving the level of security and trust through hardware-based security solutions, TC and TEE*

The advancements of cloud computing technology have facilitated the organizations and enterprises with lower cost and higher performance services that are more scalable and available. Due to these advantages, there is a high demand for cloud computing services and organizations rely on cloud technologies more and more to deliver services to their customers. However, in comparison to in-house premises, in a cloud scenario the data is stored outside the organization's perimeters. This always raises the concern about the security and privacy of the data and also trust towards the cloud provider and cloud infrastructure. This motivated us to seek for solutions that employ effective practices to secure the cloud computing infrastructure, preserve the privacy of its users, and provides trust relationship between the service provider and consumers. Such motivations are formulated as M1 and M2 in this dissertation.

In addition to cloud computing, we study the security in Internet of Things as another pervasive environment. IoT is being used in multitude of public and private sectors, ranging from the public safety to health care. This means that more and more devices are being connected to each other and to the Internet, collect and share a great deal of data including personal and business data. Security attacks on these devices could result in leaking of this data, consequently harming the business, loss of millions of dollars, and even endangering the safety and security of the consumers. The traditional software and hardware security measures that are meant for computers are not directly applicable to IoT devices and embedded systems, as they often overwhelm the limited resource and processing capability that these devices have. To these reasons, we were motivated to seek for a software security solution to improve the security of IoT devices. This led us to the third motivation of research in this dissertation (M3).

In order to secure a system, there exists a wide range of security measures to provide security in various levels, including hardware, software, and network. This dissertation investigates the security mainly in software and hardware level and it touches network level security very briefly (in Publication IV). To achieve this goal, we chose software security techniques, obfuscation and diversification, as we believe that application of these techniques at operating system and API level generates unique diversified software instances, and thus, lowers the risk of large-scale and targeted attacks. They additionally lower the risk of malicious reverse engineering of the software, make the attack more challenging, inefficient, and costly from an attacker's point of view, and have a potential to render the malware useless. This led us to research motivation M4 to study these two techniques extensively in enhancing the security of software in cloud computing and IoT. Nonetheless, software security measures might not be sufficient in some cases, therefore, additional hardware security measures make a more robust system to alleviate the risk of both software and hardware attacks. In this regard, in

addition to the two discussed software security technologies, we also use hardware based security solutions, trusted computing, and TEEs to make a more complete security approach, and increase the level of trust. This led us to research motivation M5.

3.2 Research Questions

In order to address the motivation of the studies (Section 3.1), we formulated the following research questions:

- *RQ1: How can software diversification and code obfuscation improve the security of software?*

  These two software security techniques have been extensively studied in the literature as promising techniques for protecting software systems. Through answering this question, we aim at learning what is known about the use of obfuscation and diversification in securing the software, the state-of-the-art, the types of threats these techniques could successfully mitigate, what exactly is obfuscated and diversified in order to lower the risk of such threats, when and through what methods they are applied, in what execution environments these techniques are applicable or have been applied.

- *RQ2: How can we enhance the level of security and trust in cloud computing through obfuscation, diversification and trusted computing and TEE technologies?*

  In the current scenario that many of the organizations and users are relying on cloud services, the security of it is of paramount importance. This security should be guaranteed at various levels. In addition to the significance of security, we require a trust relationship between the cloud service provider and its users. In this dissertation, we contemplate the cloud security from software security point of view, and we consider the use of trusted computing and trusted execution environments for providing trust and verifying the cloud's accountability. Through investigating this research question, we consider the possibility of applying obfuscation and diversification techniques to enhance the security and utilize trusted computing technology and trusted execution environments to guarantee the trust relationship in virtual environments and ensuring the integrity and confidentiality of the program execution.

- *RQ3: How can we enhance the level of security in IoT through obfuscation and diversification technologies?*

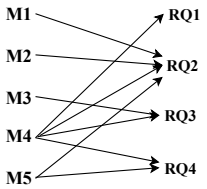
With the pervasiveness of the usage of IoT and the high amount of data collected by the IoT devices, it is very important that these devices are secured and the network on which the data is transmitted is protected. Through investigating this research question, we consider the possibility of applying obfuscation and diversification techniques to enhance the security in IoT, at network and application layer. This approach provides security at devices level, mitigates the risk of massive-scale attacks, and lowers the risk of malware.

- *RQ4: How can we combine the software security techniques, with hardware backed solutions to introduce a robust security measure?*

  It is known that the rate of malware and software attacks are increasing, thus there is a high demand for strong software protection mechanisms. However, the software security techniques have sometimes been proved to be vulnerable and breakable. On the other hand, the hardware security approaches lack the flexibility and simplicity in implementation that the software-based approaches have. Thus, by answering this research question, we aim at employing software security techniques (obfuscation and diversification) together with hardware-based security solutions (TC and TEE) to have more robust protection.

Figure 3.1 shows the relationship of research questions we formulated in this dissertation in order to follow the research motivations explained in Section 3.1.

M1: Improving security in cloud computing
M2: Improving trust in cloud computing
M3: Improving security in IoT
M4: Improving the security of software through obfuscation and diversification techniques
M5: Improving the level of security and trust through hardware-based security solutions, TC and TEE



RQ1: How can software diversification and code obfuscation improve the security of software?
RQ2: How can we enhance the level of security and trust in cloud computing through obfuscation, diversification and trusted computing and TEE technologies?
RQ3: How can we enhance the level of security in IoT through obfuscation and diversification technologies?
RQ4: How can we combine the software security techniques, with hardware backed solutions to introduce a robust security measure?

Figure 3.1: Relationship of the motivations of the research with the formulated research questions in this dissertation.

3.3 Research Methodology and Research Approach

In order to answer the defined research questions in this dissertation (RQ1 - RQ4), different research methodologies and research approaches are employed. This section presents an overview of these methodologies and approaches, and explains the type of methodology and approach that has been used in each of the original publications (Publication I - Publication VI).

Research methodology is a systematic way of solving a research problem. In a research methodology the various steps that should be adopted by a researcher are applied. Meanwhile, research methods are referred to all the techniques and methods that can be utilized in performing research operations [38]. According to the classification of the research methods presented by Wohlin and Aurum [60], besides the survey method, there are three main classes of research methods: 1) case study, 2) action research, and 3) design science research.

A *case study research* aims at investigating a phenomenon by employing multiple data collection methods and collecting information from various sources. A case study research typically follows the phases of designing, conducting the case study, analyzing the collected data, and developing a conclusion [60].

An *action research* aims at solving a problem of an organization for which the research is conducted. In this research, the researcher is part of the organization during the process of investigation, developing and applying the solution. The process of action research typically follows the phases of diagnosis of the problem, planning, action taking, evaluating, and specifying the learning [60].

A *design science research* aims at addressing a problem through building and evaluating an artefact that is designed to meet the requirements of that problem. This research is considered as a problem-solving process. This means that for solving that specific problem, an innovative and effective artefact is designed that should be evaluated by applying thorough evaluation approaches. The outcome of this research is presented in the form of a model, method, construct, or an instantiation [60].

Although there is an overlap in some stages of research approach used in action research and design science research, the focus of the outcome is different. Action research focuses on solving a problem in an organization through organizational and social changes. While, design science research focuses on bringing a solution as an IT artefact (such as a framework or a model) that will become a knowledge base for researchers.

Table 3.1 presents the research methods used in each of the original publications in this dissertation.

In Publication I, the survey research method is used, as the publication extensively observes and studies the literature and the existing solutions.

Publications	Research method
Publication I	survey
Publication II	survey & design science
Publication III	design science
Publication IV	design science
Publication V	design science
Publication IV	design science

Table 3.1: Research methods used in original publications included in this dissertation.
Publication I: Using Diversification and Obfuscation Techniques for Software Security: A Systematic Literature Review
Publication II: Obfuscation and Diversification for Securing Cloud Computing
Publication III: Security in Container based Virtualization through vTPM
Publication IV: Security in the Internet of Things through Obfuscation and Diversification
Publication V: Interface Diversification in IoT Operating Systems
Publication VI: Mitigating Branch Shadowing Attacks on Intel SGX using Control Flow Randomization

The research approach to conduct this thorough observation is systematic literature survey.

In Publication II, the survey and design science research method is used. Survey research method is used to collect and analyze the data, and draw a conclusion. More precisely, the literature is reviewed in order to identify the existing obfuscation and diversification solutions in the context of cloud security. Then, based on this conclusion, we propose an innovative method for solving the identified problem. More precisely, based on the conducted review and identified research gaps, a new software obfuscation method is proposed to enhance the security on the client-side software in cloud computing.

In Publication III, design science research method is chosen, as the previous solutions were observed (using vTPM for hypervisor-based virtual environments), and then a new model was designed and implemented to use vTPM in container-based virtualization to enhance the security in such environments. In this design science research, we design an innovative artifact in the form of a model and construct.

In Publication IV, the design science research method is used, since as the result of studying the existing security problems in IoT environments, we have proposed an innovative method to solve them, i.e., this paper proposes a solution for securing operating system and APIs of IoT devices.

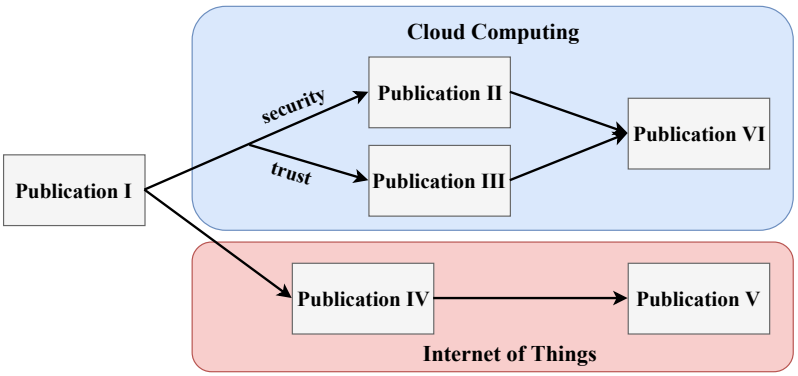
In Publication V, the engineering and design science as the research approach is used. In this paper, based on the solution proposed in our previous study (Publication IV), a new obfuscation solution is designed, developed and measured to improve the security of software on IoT devices.

In Publication VI, the design science research method is used. In this

study, we first study an existing problem (a type of side channel attack on SGX), and design a model and approach to solve this problem. This model is innovative, and the effectiveness of it is evaluated. More precisely, in this paper, a new control-flow obfuscation and randomization approach is designed, developed and measured to protect the enclave program and enhance its security.

3.4 Research Process and Publications

We started this PhD research by thoroughly studying the obfuscation and diversification techniques which was published as a Systematic Literature Review (SLR) (Publication I) [33]. As the result of this SLR study, we identified the research gaps which became directions for further research. One of the findings of this SLR was the execution environments that could benefit from the studied software security techniques, obfuscation and diversification and have not been covered in the literature. Therefore, we looked into the possibility of applying these techniques in IoT and cloud computing, as these two environments are very commonly used execution environments nowadays and there exist a vast number of security attacks on them.



Publication I: Using Diversification and Obfuscation Techniques for Software Security: A Systematic Literature Review
Publication II: Obfuscation and Diversification for Securing Cloud Computing
Publication III: Security in Container-based Virtualization through vTPM
Publication IV: Security in the Internet of Things through Obfuscation and Diversification
Publication V: Interface Diversification in IoT Operating Systems
Publication VI: Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization

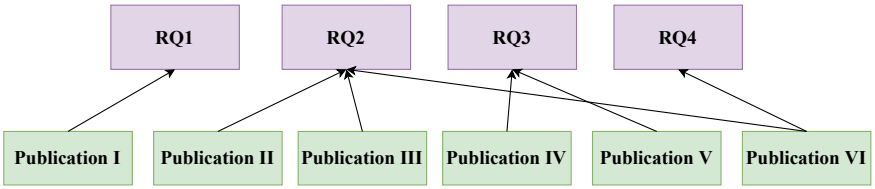
Figure 3.2: Relationship of the publications with each other in this dissertation.

Figure 3.2 shows the relation of the publications included in this dissertation. After Publication I, the research was continued in two different themes, cloud computing and Internet of Things. In the first theme we first

studied the enhancement of security in cloud through obfuscation and diversification software security techniques (Publication II), and then studied the enhancement of trust in cloud through utilizing trusted computing technology as a hardware-based approach (Publication III). Then we continued the same theme by combining these directions to provide security and trust in cloud computing through the use of obfuscation and diversification and TEE provided by Intel SGX.

The result of studies we gained in each of these publication helped us answer the research questions set in this dissertation (RQ1 - RQ4). Figure 3.3 illustrates the relationship of the publications and the research questions.

RQ1: How can software diversification and code obfuscation improve the security of software?
RQ2: How can we enhance the level of security and trust in cloud computing through obfuscation, diversification and trusted computing and TEE technologies?
RQ3: How can we enhance the level of security in IoT through obfuscation and diversification technologies?
RQ4: How can we combine the software security techniques, with hardware backed solutions to introduce a robust security measure?



Publication I: Using Diversification and Obfuscation Techniques for Software Security: A Systematic Literature Review
Publication II: Obfuscation and Diversification for Securing Cloud Computing
Publication III: Security in Container-based Virtualization through vTPM
Publication IV: Security in the Internet of Things through Obfuscation and Diversification
Publication V: Interface Diversification in IoT Operating Systems
Publication VI: Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization

Figure 3.3: Relationship of the research questions and the publications included in this dissertation.

We answered RQ1 through a comprehensive systematic literature review (Publication I). This review presented classifications of the various available obfuscation and diversification techniques, the target language and environment of obfuscation/diversification, the level of software development life-cycle in which the techniques were applied, and the types of security attacks that could be mitigated through the use of these techniques. The research gaps in this area and the potential uncovered research directions led us to our next publications.

For answering RQ2, we took two different directions: in the first one, we studied application of obfuscation and diversification in cloud computing as software-based security approaches, and in the second one, we studied the use of trusted computing and TEE technologies in cloud computing as hardware-based security approach. In the earlier direction, we surveyed how

obfuscation and diversification techniques have been previously used in cloud computing, and then we proposed and applied these techniques on them on the client-side web application (Publication II). In the latter direction, we studied the use of TPM as a trusted hardware technology and proposed the use of virtual TPM in container-based virtualization setting (Publication III). Combination of these two directions is reflected in (Publication VI) where we use both obfuscation as software-based security solution, backed with hardware-based security guarantees from Intel SGX. There, we applied control-flow obfuscation on the enclave programs of the Intel SGX architecture to mitigate the branch-shadowing side-channel attacks.

We answered RQ3 by studying the vulnerable points of IoT environments and proposed the application of obfuscation and diversification techniques for improving the security on IoT in two various layers: at application layer, by obfuscating/diversifying the operating systems and APIs of the IoT devices, and at network layer, by obfuscating/diversifying the communication protocols used among the devices (Publication IV, Publication V).

We answered RQ4 by combining the software-based and hardware-based security techniques that we had studied in the previous publications. We applied compile-time obfuscation and run-time randomization on the control flow of the Intel SGX enclave program (Publication VI).

3.5 Description of the Original Publications Included in the Dissertation

The following is a description of the set of publications included in this dissertation, to answer the formulated research questions. For each of the publications (Publication I - VI), a summary is presented that highlights the primary content of the publications, and also the contribution of author in each publication is explained.

3.5.1 Publication I: "Using Diversification and Obfuscation Techniques for Software Security: A Systematic Literature Review"

Summary: This publication presents a systematic review of the state-of-the-art of utilizing diversification and obfuscation techniques to improve the security of software. In this survey 357 papers published in the domain were collected, studied, analyzed and classified. As a result, we learned that these two software security techniques have been used in the literature for various aims and for mitigating a wide range of security attacks. We identified four broad categories of aims that the studied literature could belong to: a) making the act of reverse engineering of the program more

difficult, b) preventing the widely spreading of vulnerabilities, c) preventing unauthorized alteration of the software, and d) data hiding. Figure 3.4 illustrates the types of attacks that are successfully mitigated through using diversification and obfuscation techniques [32, 33].

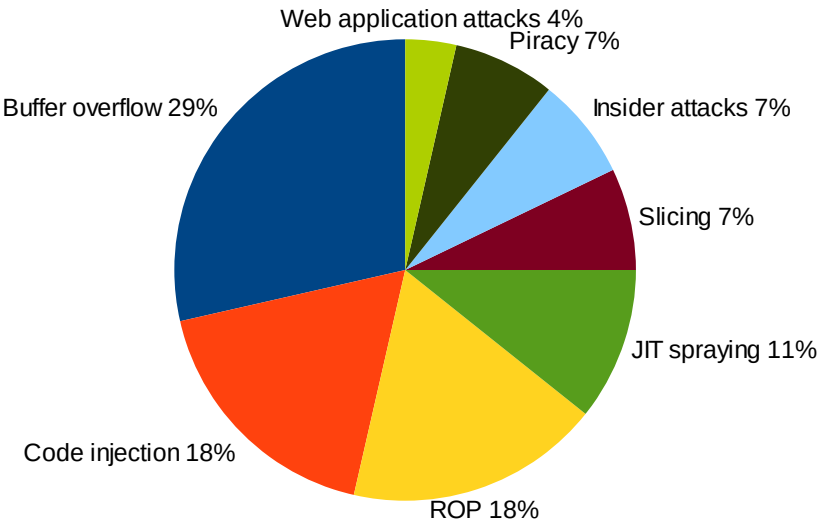


Figure 3.4: Attacks that could be thwarted using diversification and obfuscation techniques [32, 33]

Furthermore, in the literature, these techniques are proposed to secure various environments. Many of the studies claim that their approach could be applied in any environment, which reflects the broad applicability of diversification and obfuscation in different software layers and different platforms. Due to the fact that computer systems run native code on the lowest level of abstraction level, a large group of publications reported native code as the target environment for obfuscation and diversification. The other execution environments that have been discussed in smaller group of publications to benefit from these two techniques are server, cloud platform, distributed/agent based devices, mobile devices, embedded devices and desktops.

In the studied literature, obfuscation and diversification were applied on a wide range of languages to protect the program and code. A more descriptive view of the type of languages was achieved by further classifying the research into four language categories of hardware oriented, high level, scripting, and domain specific languages. Categories, the languages in each category, and number of studies considering those languages are as follows:

- Systems programming (N=158): C (52), Assembly (29), C++ (21), Cobol (1)

- Managed (N=81): Java (54), C# (3), Haskell (2), J# (1), Lisp (1), OCaml (1), VB (1)
- Scripting (N=19): JavaScript (11), Python (3), Perl (2), PHP (1)
- Domain specific, DSL (N=7): SQL (5), HTML (1)

The systems programming (C/C++) and high level languages (Java & JavaScript) represent the vast majority of the studies.

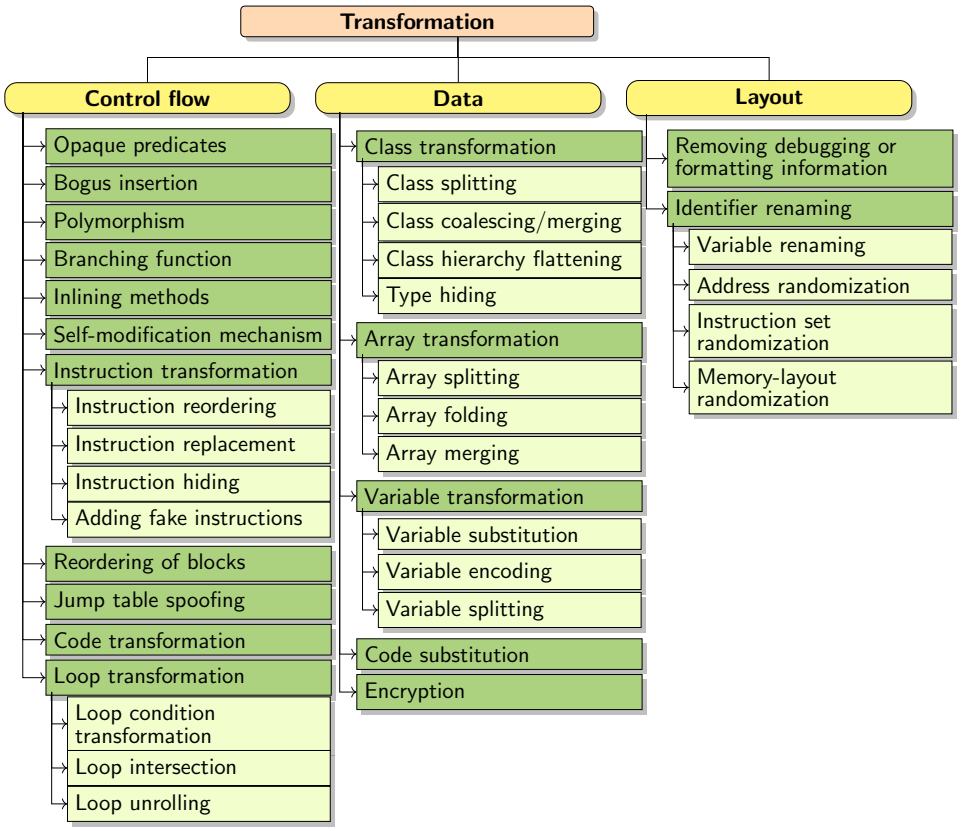


Figure 3.5: Transformation mechanisms

In the studied literature, various transformation mechanisms are proposed to make diversified program instances. Figure 3.5 illustrates these obfuscation/diversification techniques as a tree, on which the first level represents the target of the transformations and the lower levels are the transformation techniques. This classification is based on the taxonomy presented by Collberg et al. [21]. *Control flow obfuscation* obscures and alters the flow of a program in order to make it challenging for an attacker to successfully analyze and comprehend the code. To achieve this type of obfuscation, various methods have been utilized such as the use of opaque

predicates, re-ordering of blocks, and transformation of the instructions, code, and loops. *Data obfuscation* obscures data and conceals the data structure of a program. Transformation of classes, arrays, and variables are some examples of the approaches to achieve data obfuscation. *Layout obfuscation* is a type of obfuscation techniques that targets the layout structure of the program, for instance through removing debugging formatting information from the program, or identifier renaming which includes address randomization, instruction-set, and memory-layout randomization.

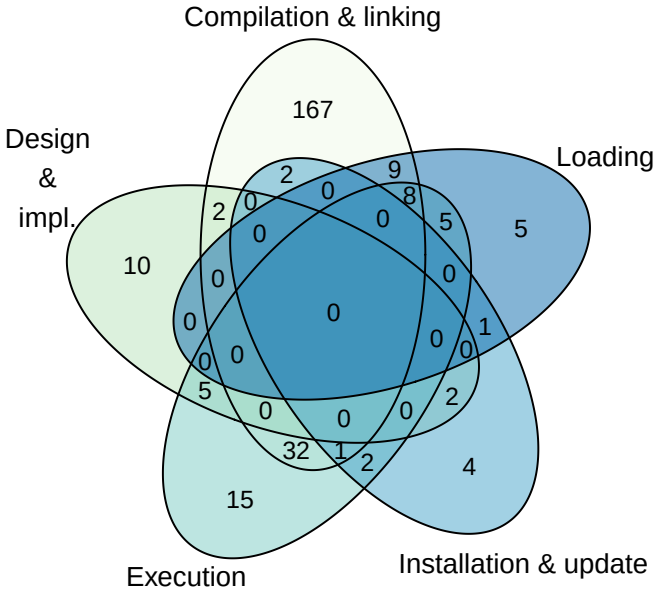


Figure 3.6: Various stages in software life-cycle that obfuscation and diversification techniques are applied on.

Each of the proposed obfuscation/diversification mechanisms is applied at different stages and levels of software development/deployment process. Figure 3.6, presents all the observed stages and their overlap in five different categories, from design to application execution time in software development life-cycle. Some of the studies applied obfuscation/diversification in a single stage, while many others operated in one to three stages. Applying obfuscation and diversification at compilation and linking time was the most commonly used phase, followed by execution time. Numbers on each stage in Figure 3.6 represent the number of studies that were applying/studying obfuscation and diversification in that particular stage.

In addition to the constructive works that were proposing a new obfuscation/diversification approach, there was a group of papers that were

empirically studying these two techniques, in the form of a survey, discussion, evaluation, optimization, experiment, comparison, and presenting a classification.

Author's contribution: In this project, the author had the primary role in leading the project, collecting and analysis of data, presenting classification and writing the paper.

3.5.2 Publication II: "Obfuscation and Diversification for Securing Cloud Computing"

Summary: This project is the result of our contribution with our colleague in Padova University, Italy. As the result of findings in Publication I, we learned that obfuscation and diversification techniques could be applied in various execution environments to boost the security. One of these environments is cloud computing. In Publication II, we first survey how obfuscation and diversification has been previously used in cloud computing to improve the security. We then propose an approach that uses these two techniques to improve the security in cloud computing environment and preserve the privacy of its users. The result of the survey (43 papers) showed that the papers in this area were employing obfuscation and diversification techniques in nine different ways to protect the cloud and its users: 1) generating noise obfuscation, 2) client-side data obfuscation as a middleware, 3) general data obfuscation, 4) source code obfuscation, 5) location obfuscation, 6) file splitting and storing on separate clouds, 7) encryption as obfuscation, 8) diversification, and 9) cloud security by virtue of securing the browser. This classification is depicted in Figure 3.7, with the number of papers in each class.

Studying the existing works shed light on the research gaps. For instance, in the majority of works in the studied set of publications, the cloud service provider is considered as untrusted and may deduce the users' data maliciously and without their consent. Therefore, they were proposing approaches that use obfuscation and diversification techniques to protect the user's data "from the cloud", and to preserve their privacy. This implies that there is room for protecting the cloud itself from the external or internal malicious parties with the help of these two techniques. This motivated us to propose a source code level obfuscation for the web applications written in JavaScript. This approach is proactive and transparent to significantly mitigate the risk of data manipulation and tampering of applications. Obfuscation alters the application that executes on the user's web browser by scrambling the HTML and JavaScript code. In this manner, it is more difficult for a malicious software to gain knowledge about the internal structure of the web application, and therefore, becomes more difficult for the attacker to compromise the application and harm it.

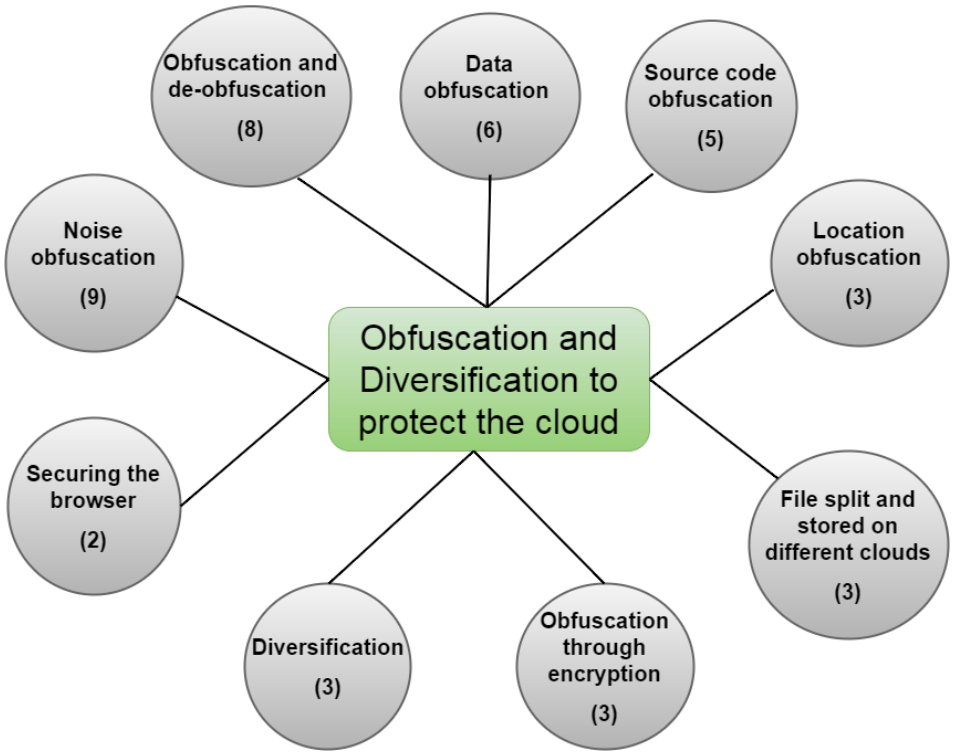


Figure 3.7: Classification of the studies that use obfuscation and diversification techniques for protecting the security and privacy in cloud computing

Author's contribution: In this project, the author had the primary role in leading the project, data collection, analysis of data, presenting classification, proposing the mitigating approach, and writing the paper.

3.5.3 Publication III: "Security in Container-based Virtualization through vTPM"

Summary: We studied integration of trusted computing technology in virtual environments and learned that in all existing works, vTPM is proposed and implemented for hypervisor-based virtualization model and no paper considers this security solution for container-based virtualization. This is while the use of containers is growing due to their benefits (e.g., being lightweight and fast). This motivated us to consider the possibility of applying vTPM in container-based virtualization. In Publication III, we propose two architectural solutions that integrate the vTPM in container-based virtualization model and extend the functionalities of TPM module to the containers. In the first design, the TPM module resides below the container-layer in the operating system kernel, and it is virtualized to make it available

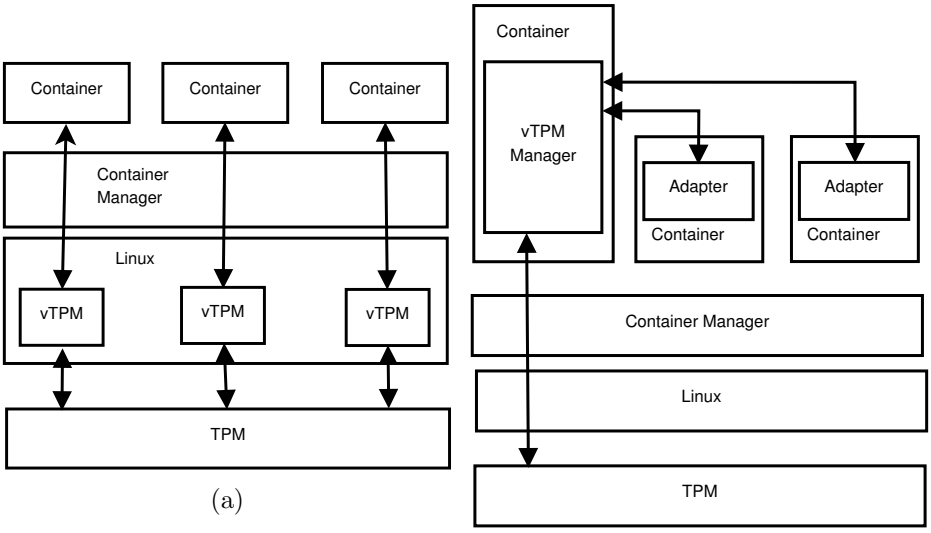


Figure 3.8: a) vTPM implemented in a kernel module, b) vTPM located in a dedicated container.

to multiple containers. In this design, a kernel module provides arbitrary number of software based vTPMs. The virtual TPMs present character device type interfaces to the userspace, and present the same interface to the software as the hardware-based TPM does. Also there is a strong association between the containers and their corresponding vTPM instance. Figure 3.8a depicts this design.

The second design is inspired by the work done by Berger et al. [16] which employs vTPM in traditional hypervisor-based virtualization model, and the vTPM manager is placed inside a separate Xen domain. Our design is proposed for container-based virtualization, and virtual TPM manager is placed in a separate container. vTPM management container has access to the hardware TPM and exposes vTPM interface to other containers through a communication channel which could be a local UNIX domain socket or another Inter-Process Communication (IPC) mechanism. Figure 3.8b depicts the second architectural design in Publication III.

Author's contribution: In this project, the author had the primary role in studying the use of trusted computing for securing virtual environments and co-contributed in proposing, designing and implementing the architectures for using the vTPM in container-based virtualization setting. The author had the primary role in writing and the paper.

3.5.4 Publication IV: "Security in the Internet of Things through Obfuscation and Diversification"

Summary: This publication studies the vulnerabilities and the existing weak points that challenge the security of IoT devices. The IoT devices range from more potent 32-bit processors (e.g., smart phones) to the very lightweight sensors controlled by 8-bit micro-controllers with limited computational power and memory. Thus, the chosen software for these devices should be applicable to a wide range of devices, including the lowest power ones and capable of supporting the functionality of the object. Such a scenario, due to the special characteristics of these devices, makes the security measures fairly limited. IoT sensors and devices contain embedded chips that function with the help of an operating system and APIs, which could be prone to malware and software attacks, and hence need to be protected. The first proposed idea in Publication IV is on applying obfuscation and diversification techniques on the operating systems and APIs on the IoT devices to protect them at the application layer. Obfuscation makes the IoT software harder to reverse engineer and be accessed, whereas diversification thwarts the massive-scale attacks. This means that if the attacker manages to attack one device through designing an attack model, she cannot take other devices under control with the same attack model, because they have differently diversified and obfuscated software, although with the equivalent functionality. This is a very significant outcome in the highly distributed systems to lower the risk of massive scale attacks occurring.

The second proposed idea in Publication IV is applying these techniques on some communication protocols to protect the devices at the access protocol level. In a communication network, an application level protocol defines the interfaces and the shared protocols used by the communication parties. Protocol identification is the act of identifying what protocol is used in the communication session, and it can be done via static analysis methods and comparing the protocol used in the communication with the common existing protocols. The information gained from this analysis could be used by an intruder and could endanger the confidentiality and integrity of the communication. Protocol obfuscation could protect them from being identified and make them more difficult to be recognized by the traffic classification machines. Obfuscation removes the identifiable properties from the protocol, e.g., packet size and byte sequence and make them look random [28]. We propose obfuscating the communication protocol among a small set of nodes (e.g., within a home) in a way that the obfuscation method is kept secret among them and only the nodes that know the secret are able to communicate with each others. By changing/complicating the form of the protocol and making it different from the default format, we aim at generating a huge number of unique diversified protocols from a reference protocol.

Besides the protocol obfuscation, we propose protocol diversification, which considers the protocol as an operation of two state machines, so that (synchronized) state changes are messages sent between parties. The original implicit state machine of a protocol can be diversified by adding/splitting new states and transitions.

Author's contribution: In this project, the author had the primary role in studying the security status of IoT devices, proposing the idea of using obfuscation and diversification to secure these environments (at application and network layer), and writing the paper.

3.5.5 Publication V: "Interface diversification in IoT operating systems"

Summary: This publication follows the work done in Publication IV, and applies diversification on the interfaces of IoT operating systems. More precisely, we apply diversification in post-compilation and linking phase of the software life-cycle, by shuffling the order of the linked objects while preserving the semantics of the code. This approach successfully prevents malicious exploits from producing adverse effects in the system. Besides shuffling, we also apply the library symbol diversification method, and construct the required support for it, for example into the dynamic loading phase. Besides studying and discussing memory layout shuffling, and symbol diversification as a security measure for IoT operating systems, we provide practical implementations for these schemes for the Thingsee and Raspbian operating system and test these solutions to show the feasibility of diversification in IoT environments.

We applied layout diversification on Thingsee operating system with a flat address-space configuration. The main goal was to prevent Return-Oriented Programming (ROP) attacks that depend on known entrypoint addresses, from functioning. First, we wrote a vulnerable Thingsee operating system application program that writes user-supplied data to a stack buffer without bounds-checking. We devised an exploit that overwrites a return address on the stack and makes the program jump to a chosen function. After executing the function, the system apparently crashes and reboots. Thence, we rebuilt the Thingsee operating system using our modified GNU ld linker with randomization of the order of functions and data in the Thingsee operating system image. An identical exploit did not cause the chosen function to be executed this time, but it nevertheless invoked undefined behavior and, in our case, made the operating system crash. Our vulnerable program was simply created by modifying the "Hello World" example in the Thingsee operating system in order to facilitate creating a new Thingsee operating system application. Our vulnerable program takes an input from a command-line argument as a hexadecimal string and writes it

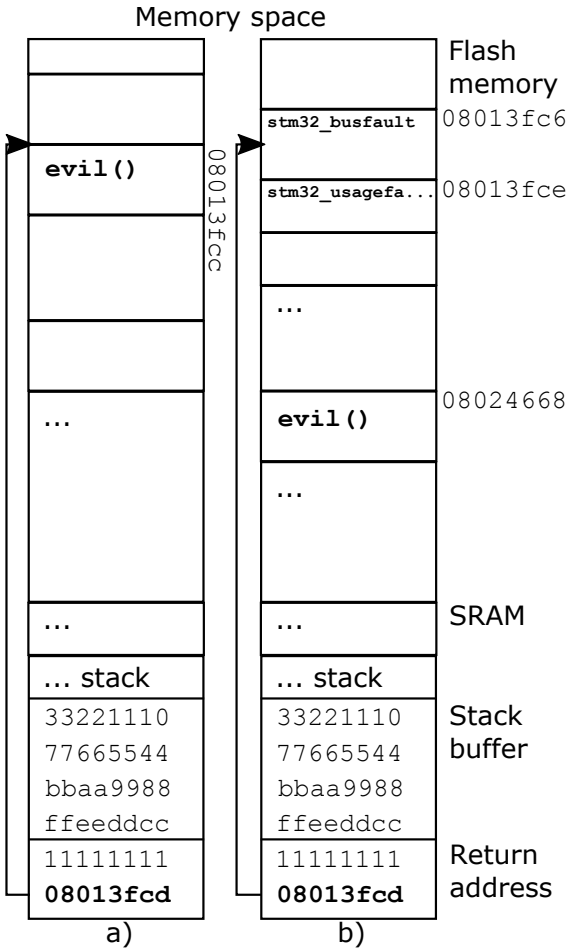


Figure 3.9: a) Normal operating system: execution jumps to beginning of evil(), b) operating system with diversified layout: execution jumps to the middle of an instruction.

to a stack buffer in a hex-decoded form without bounds-checking. We use this method of input to simulate a malicious input that would be received from a real application environment, such as a network server. Figure 3.9 illustrates how execution flows in a normal operating system and an operating system with diversified layout. In the diversified operating system the execution jumps to an address that is no longer the address to the malicious function, so the attacker cannot execute its code.

The second part of the work done in this publication is a diversification scheme we implemented for Raspbian on a Raspberry Pi device. In order to utilize critical resources of a device, typically applications use the well-known libraries. Diversifying the symbol names in these libraries, makes

it impossible for the adversary to use the well-known symbol names of the operating system libraries and attack the system. The diversification needs to be propagated to the legitimate applications, so they could still access the libraries. As a result, the diversified applications know some part of the diversification secret (some set of the diversified symbol names) and are compatible with the library binaries that include the important functions. In addition to library functions, some IoT operating systems support system calls for providing services to user programs. In such setting, the system call diversification should be applied together with symbol diversification. For performing this diversification, a symbol diversification tool was used that we had previously designed and build for diversifying x86 64 Linux [39]. The tool was modified to support 32-bit ELF files.

Author's contribution: In this project, the author had the role of advising in the choice of diversification method and the APIs to apply diversification on, based on the thorough systematic literature review that was done previously on the topic (Publication I). The author also participated in writing and presenting the paper.

3.5.6 Publication VI: "Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization"

Summary: This publication was the result of author's research visit in Secure System Lab in Aalto University, Finland. We studied the branch-shadowing attack as one of the side-channel attacks on Intel SGX and proposed and implemented a defensive solution to thwart this attack. We applied program control flow obfuscation and run-time code randomization in Intel SGX environment, with the aim of protecting the enclave program's control flow.

Branch-shadowing attack is one of the side channel-attacks on Intel SGX. The main goal of this attack is to infer the control flow of the program running inside an enclave. Figure 3.10 shows a simple example of a case of a conditional branch that has been taken. The blue line in Figure 3.10 shows that the branch is taken. The attacker gains the source code or binary of the victim program through static/dynamic analysis and designs a shadow of this code which is aligned with the victim's code in terms of branches and their target addresses. Then the malicious operating system that has control over the execution of the enclave, interrupts the enclave execution and gives the control to the shadow code. There, Last Branch Record (LBR) is enabled which reports the prediction hits and misses. The shadow code then starts executing. It gets to Block0' and Branch Target Buffer (BTB) predicts that the branch should be taken, and because the jumps in Block0 and Block0' are aligned, what had happened inside enclave, affects the prediction of this jump on the shadow code. So in the attack code the jump to Block2' is taken as well. Then the attacker reads LBR that reports the hits and misses of

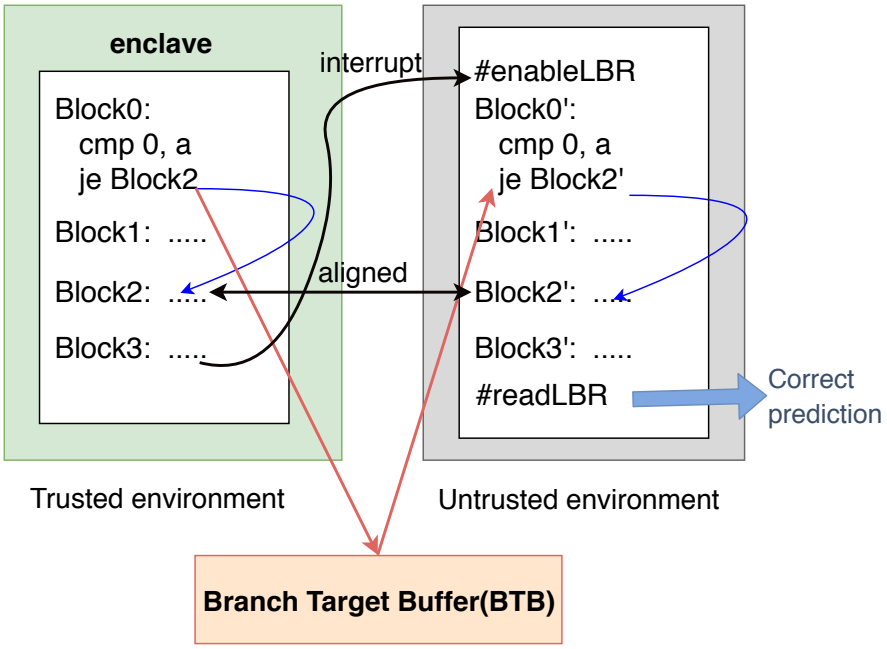


Figure 3.10: Branch shadowing attack

branch predictions, and learns that the prediction has been correct, meaning that the conditional branch has been taken. In this way, the value of "a" is leaked. Since the control flow of the program is dependent on the value of "a", this means that the control flow of the program running inside the enclave is also leaked to the attacker.

To defend against this attack, we proposed and implemented a novel approach that is composed of two main components: an obfuscating compiler and a run-time randomizer. The obfuscating compiler modifies the code by converting all branching instructions to indirect branches. We use conditional moves as replacements for conditional branch. Trampolines are also created at this point. Trampolines are minimal code sections that include intermediate jumps/bounces to the target locations. We assume that code and trampolines are known by the attacker. Trampolines are randomized inside the enclave at run-time by the randomizer to prevent the attacker from reliably tracking their execution. Randomizing the layout of the trampoline, forces the attacker to shadow all possible locations. The finite size of the BTB limits the number of guesses the attacker can perform, and thus we can quantify and limit the success probability of a branch-showing attack using the size of the trampoline as a tunable security parameter. Figure 3.11 illustrates the high level design of our system, and also depicts how the obfuscating compiler modifies the code at compile time.

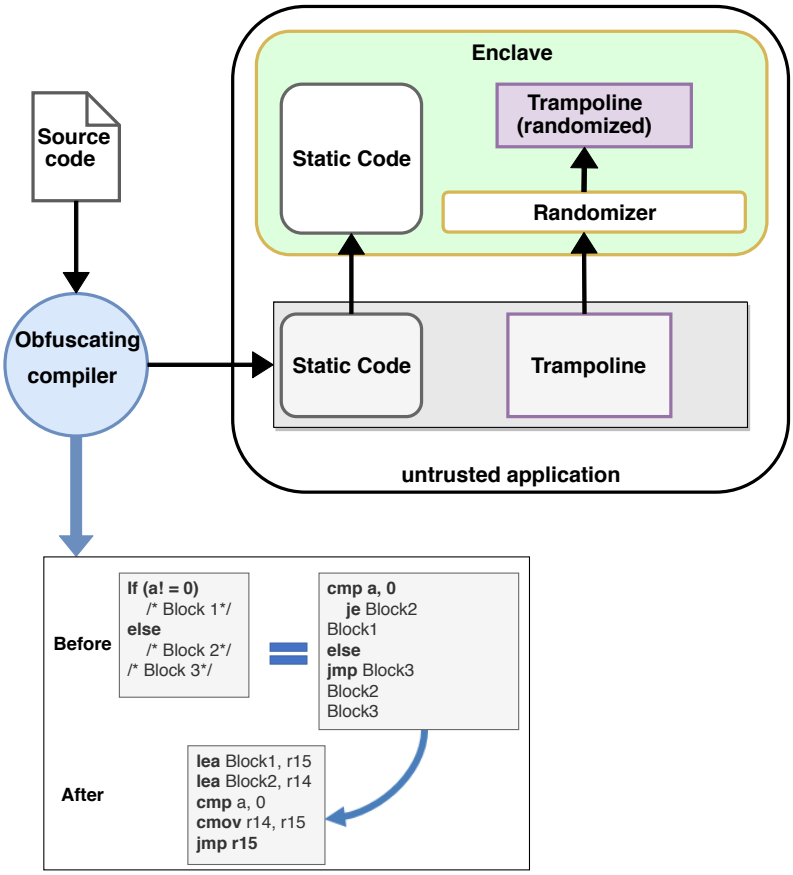


Figure 3.11: Our proposed approach for mitigating branch shadowing attack.

Author's contribution: In this project, the author had the primary role in proposing the idea, co-contribution in design, implementation, benchmarking the system, writing and presenting the paper.

Chapter 4

Conclusions

The advancement in cloud computing technology and the beneficial services and resources that it offers to the consumers, have made it a very popular technology in the past decade. The services offered by the cloud include infrastructure for processing and storage of data, platform for software development, and software to serve the application-level business needs. The profound reliance of the enterprises and businesses on cloud makes it more and more significant to employ robust security measures to protect the infrastructure from insider and outsider threats.

The other widely used technology is IoT, which is in use in various sectors to provide services ranging from healthcare and wearable data monitoring systems, to smart lighting, grids and farming. The high amount of data these devices collect raises security and privacy concerns, as some of the collected and shared data over the network might contain personal or critical data. The instances of security incidents in the past years have shown that security of IoT still needs more attention from developers and researchers.

The widespread use of cloud computing and IoT and the current security status in these environments motivated us to focus this PhD research on improving the level of security and trust in these execution environments. To achieve this goal, we used two software security techniques, obfuscation and diversification, and two hardware-based security solutions, trusted computing and TEE. By obfuscation and diversification of the operating systems and APIs of the IoT devices, we secure them at the application level, and by obfuscation and diversification of the communication protocols, we protect the communication of data between them at the network level. For securing cloud computing, we looked into the use of obfuscation and diversification for securing the cloud computing software at the client-side. Moreover, we studied the hardware based security techniques, TPM and SGX, for providing higher level of security and layered trust in various layers from hardware to the application.

This PhD dissertation was organized in 4 different chapters. Chapter 1 presented the introductory remarks, and Chapter 2 presented fundamental background on the concepts discussed in this dissertation. Chapter 3 presented in detail the research conducted in this dissertation. This chapter (Chapter 4) presents an overview of contribution made by this dissertation, limitations of the study and future directions for further research.

4.1 Contributions

The following revises the contribution made by this dissertation (C1 - C4) to the security and trust in the context of cloud computing and IoT.

In the following, we explain each of the contributions in detail and discuss how we answered the research questions and delivered the contributions through the different original publications included in this dissertation:

- *C1: Obfuscation and diversification for improving the security of software:* The two techniques have been extensively used in the literature through various mechanisms to improve the security of software. Publication I contributes to C1 and answers the *RQ1: How can software diversification and code obfuscation improve the security of software?* by collecting and studying all the studies published in the domain. Our contribution in this domain was to present classification of these studies in respect to various perspectives, such as the obfuscation and diversification mechanisms used, the stages of application in software development life-cycle, target of application, languages, and the execution environments. We highlighted the aims of use of these two software security techniques and the security attacks successfully mitigated by them. Furthermore, we pinpointed the research gaps, limitations, and proposed solutions and future research directions.

- *C2: Improving security and trust in cloud computing:* In order to consolidate the security in cloud computing, we chose software security solutions (obfuscation and diversification) backed with hardware based solutions (TPM and TEE). In this respect, Publication II studied how obfuscation and diversification have previously been used to enhance the security in cloud, and proposes a client-side software security obfuscation and diversification technique. Publication III proposes and implements the use of vTPM in container-based virtualization to increase the level of security and trust in this virtual environment. Publication VI also contributes to both security and trust in cloud computing through the studying of Intel SGX as a technology to provide secure and trustworthy execution environment. More specifically, in this line of research, Publication VI studies one of the side-channel

attacks on Intel-SGX and proposes and implements a practical solution to mitigate this attack. Publication II and Publication III answer *RQ2: How to enhance the level of security and trust in cloud computing through obfuscation, diversification and trusted computing technologies?*, whereas Publication VI answers to *RQ4: How can we combine the software security techniques, with hardware backed solutions to introduce a robust security measure?*

- *C3: Improving security in IoT:* In order to consolidate the security in IoT, we considered obfuscation and diversification as two potential techniques to secure IoT devices. Publication IV conduces to this contribution by proposing the application of obfuscation and diversification on the operating systems of the IoT devices and on the communication protocols used by these devices to protect them at network and application level. Publication V contributes to C3 by applying memory layout shuffling, and symbol diversification on the APIs of operating systems on IoT devices. These approaches make the malware ineffective and unable to execute its code, and also prevent the massive-scale attacks on the IoT devices.

- *C4: Combining the software security techniques with hardware backed solutions:* As discussed before, the security solutions that are merely software-based might not be always the best practices and the strongest defense mechanisms. To this end, combining the software-based security approached with hardware-based solutions reinforces the overall security level of the system (RQ4). In this regard, Publication VI contributes to C4 and answers RQ4 by utilizing obfuscation as the software security technology and Intel SGX as the hardware-based security solution. More precisely, in Publication VI we apply compile-time obfuscation and and run-time code randomization on the enclave program's control flow.

We believe that the outcome of this thesis could have tremendous results for future academic research in the field, as well as industry. Combining the software and hardware techniques makes a more complete and robust security approach for different execution environments including cloud and IoT that we studied extensively in this thesis.

The application of the proposed approaches in this thesis mitigates large-scale attacks. Due to the identical design of the IoT devices, the developers and vendors of these devices could greatly benefit from the diversification mechanisms proposed in this thesis to diversify the operating systems and APIs to prevent breach of all devices, if one device was breached.

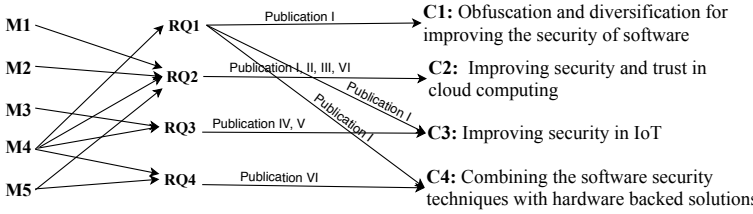
The proposed software security approaches in this thesis do not aim at removing security holes, but by making it difficult (or at best impossible) to

access and use these security holes. Moreover, these approaches reduce the reliance on the malware scanners, as they do not rely on detecting the malware, but they aim at preventing the malware from performing its malicious activities and thus, making it effective. These results have very positive impact in the software security business.

Also, since Intel has explicitly stated that the side channel attacks on SGX is out of the scope of their support, our proposed solution to mitigate the branch shadowing channel attacks on Intel SGX brings a significant value to the research in the field and also for the consumers of the product.

Figure 4.1 illustrates the relationship of the motivations of the research, research questions, and the contributions made in this dissertation through the delivered publications. M1 - M5 are the motivations of research in the domain, security and trust in cloud computing and IoT (discussed in detail in Chapter 3), from which M1 - M3 are the motivations associated with the choice of execution environment, and M4 - M5 are the motivations associated with the choice of security approach. To follow the stated motivations we formulated the research questions RQ1 - RQ4. The result of the conducted research is published in six different publications that are included in this dissertation, Publication I through Publication VI. The collection of these publications provides the set of contributions C1 - C4.

M1: Improving security in cloud computing
M2: Improving trust in cloud computing
M3: Improving security in IoT
M4: Improving the security of software through obfuscation and diversification techniques
M5: Improving the level of security and trust through hardware-based security solutions, TC and TEE



RQ1: How can software diversification and code obfuscation improve the security of software?
RQ2: How can we enhance the level of security and trust in cloud computing through obfuscation, diversification and trusted computing and TEE technologies?
RQ3: How can we enhance the level of security in IoT through obfuscation and diversification technologies?
RQ4: How can we combine the software security techniques, with hardware backed solutions to introduce a robust security measure?

Figure 4.1: Relationship of the motivations, research questions, and the contributions made through publications included in this dissertation.

4.2 Challenges

Interface diversification is a potential approach to generate unique versions of software instances in a way that each machine has its own version which is differently modified from other machines. This different modification makes the diversified interfaces incompatible with the ones that are not diversified or diversified differently. This feature, on one hand, has the benefit that it makes a piece of malware incompatible and makes it unable to communicate with the environment (e.g., call the system calls) and execute its code. On the other hand, it raises the challenge about managing the diversification secret (i.e., securing and propagation of it). The following are the challenges (Ch1 - Ch6) we identified in this field of research for applying obfuscation and diversification for cloud and IoT:

- *Ch1- Propagation of the diversification secret*: all the legitimate applications/interfaces that we would like to maintain their access to the diversified machine need to be also diversified the same way and with the same diversification secret so they could be compatible. These legitimate applications need to know how and using what identifier they could access the resources on the diversified machine. Propagating the diversification secret is still a challenge that needs to be managed.

- *Ch2- Securing the diversification secret*: the information regarding the diversified interfaces and how to interact with them needs to be kept secret from illegitimate applications. This prevents a malware from using them and access the system resources. In many cases such as diversifying the system call or shared libraries, this secret is in the form of a key or a hash value that could be utilized to transform the original interfaces into diversified ones. The way and the place to securely store this secret is another challenge for interface diversification.

- *Ch3- Updating the diversified software*: Patches and updates are very essential to protect the software. Managing the update of the diversified software is still an open issue. In ideal case, each software could have diversification built-in so that it could be configured with the diversification secret. At this moment, this is not feasible and at the time of each update, the software needs to be recompiled manually.

- *Ch4- Interface diversification of the IoT devices*: When studying the application of diversification on the interfaces of IoT devices and sensor networks, we learned that some of these devices have an operating system (e.g., RIOT and TinyOS) that has limited number of diversifiable interfaces. Meanwhile, some others (e.g., Linux and Android Things) have several essential interfaces that could be diversified and

benefit from the protection this technique offers. In addition, some of these operating system are not fully open-source and the source code is not completely available. This, introduces challenge to propagation of diversification secret throughout the whole system.

- *Ch5- Associated overhead and cost*: Using diversification and obfuscation is always a race against time, in a way that the adversary potentially could guess the diversification secret correctly or de-obfuscate the code. Dynamically obfuscating/diversifying is an adequate defense, but raises complexity and performance costs of the approach and is difficult to implement.

- *Ch6- Non-compliant TPMs*: One unresolved issue is trusted computing technology us the lack of standardization through which the heterogeneous devices could interact with each other. This means that TCG does not require certain set of implementation specifics to its adopters. As a result, some of the available TPMs do not comply with TPM specifications, and this will be the case for the foreseeable future. This will bring along the limitation and challenge for the future interoperability of different trusted platforms [15].

4.3 Future Directions

In this PhD dissertation various aspects of software and hardware security were discussed and studied, as a result of which various publications were published tackling the security and trust concepts in cloud computing and IoT. While we argue that this dissertation work presents a valuable contribution to the present day security of IoT and cloud computing, we consider the following as the future research directions concerning the dissertation's studied topics:

vTPM diversification and obfuscation: regarding the work done in Publication III on virtualizing the TPM to protect virtual environments, we consider applying obfuscation on the vTPM software to make it more difficult to break, and apply diversification in order to generate unique vTPM instances. In such a scenario, each of the containers/VMs receives a uniquely diversified version of the vTPM software. In the case that one container/VM is compromised and the malware manages to access the vTPM and executes its code, the same malware cannot work on other vTPM instances. Containers themselves could also benefit from diversification and obfuscation approaches. Interfaces and APIs of containers could potentially be diversified so that each container would have a diverse execution environment.

Secure storage of the diversification secret: as discussed earlier (in Section 4.2), the diversification secret needs to be propagated to the trusted

APIs and applications, and kept secret from the untrusted ones to prevent their access to the system resources. Considering that TPM offers a secure storage for storing cryptographic keys, it could also be a potential place to hold the diversification secret.

PKI system for diversification: basic version of diversification is based on secret key cryptography. One valuable line of research would be to find whether there could be any feasible interpretation of diversification based on public key cryptography. This means that the public key is used for diversification and shared publicly, and the private key is used by execution engine(s) to de-diversify the diversified program, in order to be able to execute it. In this scenario also, TPM could be used to hold the private key.

Mitigation of a wider range of side-channel attacks on Intel SGX: regarding the work done in Publication VI, we plan to continue this line of research by integrating our approach with other defence mechanisms available such as T-SGX [54], SGX-Shield [52] in order to mitigate a wider range of side-channel attacks.

Bibliography

[1] Cloud security alliance (CSA). https://cloudsecurityalliance.org/. Verified 2019-02-12.

[2] Extensible Messaging and Presence Protocol (XMPP). http://tools.ietf.org/html/rfc6121. Verified 2019-02-15.

[3] Intel® SGX and Side-Channels. https://software.intel.com/en-us/articles/intel-sgx-and-side-channels. Verified: 2019-10-13.

[4] Intel® Software Guard Extensions (Intel® SGX). https://software.intel.com/en-us/sgx. Verified: 2019-02-22.

[5] Intel® Software Guard Extensions Programming Reference, Revision 2. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. Verified: 2019-02-22.

[6] Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025. https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/. Verified: 2020-09-06.

[7] Internet of Things research study—HP report. https://www8.hp.com/us/en/hp-news/press-release.html?id=1744676. Verified 2019-02-15.

[8] IPv6 over Low power WPAN (6LowPAN). https://tools.ietf.org/wg/6lowpan/. Verified Verified 2019-02-15.

[9] nesC: A Programming Language for Deeply Networked Systems. http://nescc.sourceforge.net/. Verified 2019-02-14.

[10] Online C obfuscator. https://picheta.me/obfuscator/. Verified: 2019-03-12.

[11] Owasp Internet of Things project. {https://www.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=OWASP_Internet_of_Things_Top_10_for_2014}, note = Verified 2019-02-15.

[12] Trusted Computing Group (TCG). http://www.trustedcomputinggroup.org/. Verified: 2019-02-21.

[13] Trusted Platform Module (TPM). http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module. Verified: 2019-02-21.

[14] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas Schmidt. OS for the IoT-goals, challenges, and solutions. In *Workshop Interdisciplinaire sur la Sécurité Globale (WISG2013)*, page 7 pages, 2013.

[15] S. Balfe, E. Gallery, C. J. Mitchell, and K. G. Paterson. Challenges for trusted computing. *IEEE Security & Privacy*, 6(6):60–66, 2008.

[16] Stefan Berger, Ramón Cáceres, Kenneth Goldman, Ronald Perez, Reiner Sailer, and Leendert Doorn. vTPM: Virtualizing the trusted platform module. In *USENIX Security*, pages 305–320, 2006.

[17] Indranil Bose and Raktim Pal. Auto-id: Managing anything, anywhere, anytime in the supply chain. *Commun. ACM*, 48(8):100–106, August 2005.

[18] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR.SGX: hardening SGX enclaves against cache attacks with data location randomization. *CoRR*, abs/1709.09917, 2017.

[19] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.

[20] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009.

[21] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[22] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998.

[23] Hoang T. Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[24] Stephen Drape, Oege de Moor, and Ganesh Sittampalam. Transforming the .net intermediate language using path logic programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '02, pages 133–144, NY, USA, 2002. ACM.

[25] Rajdeep Dua, A. Reddy Raja, and Dharmesh Kakadia. Virtualization vs Containerization to Support PaaS. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614, March 2014.

[26] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pages 455–462, Nov 2004.

[27] Kazuhide Fukushima, Shinsaku Kiyomoto, and Toshiaki Tanaka. Obfuscation mechanism in conjunction with tamper-proof module. In *Computational Science and Engineering. CSE '09. International Conference on*, volume 2, pages 665–670, 2009.

[28] Erik Hjelmvik and Wolfgang John. Breaking and improving protocol obfuscation. *Chalmers University of Technology, Tech. Rep*, 123751, 2010.

[29] Shohreh Hosseinzadeh, Sami Hyrynsalmi, Mauro Conti, and Ville Leppänen. Security and privacy in cloud computing via obfuscation and diversification: A survey. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)(CLOUDCOM)*, volume 00, pages 529–535, Nov. 2016.

[30] Shohreh Hosseinzadeh, Samuel Laurén, Sampsa Rauti, Sami Hyrynsalmi, Mauro Conti, and Ville Leppänen. Obfuscation and diversification for securing cloud computing. In Victor Chang, Muthu Ramachandran, Robert J. Walters, and Gary Wills, editors, *Enterprise Security*, pages 179–202, Cham, 2017. Springer International Publishing.

[31] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating branch-shadowing attacks on intel sgx using control flow randomization. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, SysTEX '18, pages 42–47, New York, NY, USA, 2018. ACM.

[32] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. A survey on aims and environments of diversification and obfuscation in software security. In *Proceedings of the 17th International Conference on Computer Systems and Technologies 2016*, CompSysTech '16, pages 113–120, New York, NY, USA, 2016. ACM.

[33] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, 104:72 – 93, 2018.

[34] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. MQTT-S–A publish/subscribe protocol for Wireless Sensor Networks. In *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, pages 791–798, Jan 2008.

[35] Chongkyung Kil, Jinsuk Jim, C. Bookholt, J. Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference. ACSAC '06. 22nd Annual*, pages 339–348, Dec 2006.

[36] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *CoRR*, abs/1801.01203, 2018.

[37] Aki Koivu, Lauri Koivunen, Shohreh Hosseinzadeh, Samuel Laurén, Sami Hyrynsalmi, Sampsa Rauti, and Ville Leppänen. Software security considerations for IoT. In *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 392–397, Dec 2016.

[38] Chakravanti Rajagopalachari Kothari. *Research methodology: Methods and techniques*. New Age International, 2004.

[39] S. Laurén, P. Mäki, S. Rauti, S. Hosseinzadeh, S. Hyrynsalmi, and V. Leppänen. Symbol diversification of linux binaries. In *World Congress on Internet Security (WorldCIS-2014)*, pages 74–79, Dec 2014.

[40] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. *CoRR*, abs/1611.06952, 2016.

[41] Phil Levis, Samuel Madden, J. Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for sensor networks. In W. Weber, J. M. Rabaey, and E. Aarts, editors, *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[42] Zhaohui Liang, Bin Liang, and Lupin Li. A system call randomization based method for countering code-injection attacks. In *International Conference on Networks Security, Wireless Communications and Trusted Computing, NSWCTC*, pages 584–587, 2009.

[43] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *CoRR*, abs/1801.01207, 2018.

[44] Kangjie Lu, Wenke Lee, Stefan Nürnberger, and Michael Backes. How to make ASLR win the clone wars: Runtime re-randomization. In *NDSS*, San Diego, CA, USA, 2016. 2016 Internet Society.

[45] Tim Mather, Subra Kumaraswamy, and Shahed Latif. *Cloud security and privacy: an enterprise perspective on risks and compliance.* " O'Reilly Media, Inc.", 2009.

[46] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Intel Corporation*, 2013.

[47] Peter Mell and Tim Grance. The NIST definition of cloud computing. page 7, 2011. National Institute of Standards and Technology Special Publication 800-145.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[49] Jon Postel. User Datagram Protocol (No. RFC 768). Technical report, 1980.

[50] Sampsa Rauti, Lauri Koivunen, Petteri Mäki, Shohreh Hosseinzadeh, Samuel Laurén, Johannes Holvitie, and Ville Leppänen. Internal interface diversification as a security measure in sensor networks. *Journal of Sensor and Actuator Networks*, 7(1), 2018.

[51] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, Aug 2015.

[52] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling address space layout randomization for sgx programs. In *Network and Distributed System Security Symposium*, 2017.

[53] Zach Shelby, Klaus Hartke, and Carsten Bormann. The constrained application protocol (CoAP). 2014.

[54] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, 2017.

[55] Ed Skoudis and Lenny Zeltser. *Malware: Fighting malicious code*. Prentice Hall Professional, Upper Saddle River, NJ 07458, 2004.

[56] Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. Obfuscation of design intent in object-oriented applications. In *Proceedings of the 3rd ACM Workshop on Digital Rights Management*, DRM '03, pages 142–153, New York, NY, USA, 2003. ACM.

[57] J.C. Spradlin. Security through opcode randomization, June 21 2012. US Patent App. 12/972,433.

[58] Steve Vinoski. Advanced Message Queuing Protocol. *Internet Computing, IEEE*, 10(6):87–89, Nov 2006.

[59] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, VA, USA, 2000.

[60] Claes Wohlin and Aybüke Aurum. Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering*, 20(6):1427–1455, Dec 2015.

[61] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015.

[62] Raghu Yeluri and Enrique Castro-Leon. Attestation: Proving trustability. In *Building the Infrastructure for Cloud Security*, pages 65–91. Springer, 2014.

[63] Raghu Yeluri and Enrique Castro-Leon. *Platform Boot Integrity: Foundation for Trusted Compute Pools*, pages 37–64. Apress, Berkeley, CA, 2014.

[64] I. You and K. Yim. Malware obfuscation techniques: A brief survey. In *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, Nov 2010.

Chapter 5

Original Publications

The following original publications are reprinted with the permissions of the respective publishers.

Publication I

Diversification and Obfuscation Techniques for Software Security: a Systematic Literature Review

Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, Ville Leppänen, In Journal of Information and Software Technology (IST), 104, 72 – 93, Elsevier, 2018.

Information and Software Technology 104 (2018) 72–93

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Diversification and obfuscation techniques for software security: A systematic literature review

Shohreh Hosseinzadeh*,[a], Sampsa Rauti[a], Samuel Laurén[a], Jari-Matti Mäkelä[a], Johannes Holvitie[a], Sami Hyrynsalmi[b], Ville Leppänen[a]

[a] Department of Future Technologies, University of Turku, Vesilinnantie 5, Turku 20500, Finland
[b] Laboratory of Pervasive Computing, Tampere University of Technology, Pohjoisranta 11 A, Pori 28100, Finland

ARTICLE INFO

Keywords:
Diversification
Obfuscation
Software security
Systematic literature review

ABSTRACT

Context: Diversification and obfuscation are promising techniques for securing software and protecting computers from harmful malware. The goal of these techniques is not removing the security holes, but making it difficult for the attacker to exploit security vulnerabilities and perform successful attacks.

Objective: There is an increasing body of research on the use of diversification and obfuscation techniques for improving software security; however, the overall view is scattered and the terminology is unstructured. Therefore, a coherent review gives a clear statement of state-of-the-art, normalizes the ongoing discussion and provides baselines for future research.

Method: In this paper, systematic literature review is used as the method of the study to select the studies that discuss diversification/obfuscation techniques for improving software security. We present the process of data collection, analysis of data, and report the results.

Results: As the result of the systematic search, we collected 357 articles relevant to the topic of our interest, published between the years 1993 and 2017. We studied the collected articles, analyzed the extracted data from them, presented classification of the data, and enlightened the research gaps.

Conclusion: The two techniques have been extensively used for various security purposes and impeding various types of security attacks. There exist many different techniques to obfuscate/diversify programs, each of which targets different parts of the programs and is applied at different phases of software development lifecycle. Moreover, we pinpoint the research gaps in this field, for instance that there are still various execution environments that could benefit from these two techniques, including cloud computing, Internet of Things (IoT), and trusted computing. We also present some potential ideas on applying the techniques on the discussed environments.

1. Introduction

In most organizations, information is a key asset that comes in the form of, for example, financial information, client data, and product design data. Intentional or accidental leakage of any of this information exposes both the business and the customers. Therefore, it is highly significant for any business to have security strategies for protecting the information and services and ensuring the confidentiality, integrity, and availability of the information.

Computer security assures that the system functions under the expected circumstances, and prevents undesired behavior. Many security breaches begin with identifying and exploiting the vulnerabilities in the system. Vulnerabilities are the defects that occur in the process of design and implementation of the software. Defects in design are known as flaws, and the defects in implementation are known as bugs. To ensure the security of software, we need to prevent or mitigate the risk of software vulnerabilities. In other words, we should either eliminate these bugs and flaws, or make it harder to exploit them.

In this paper, we *focus on making exploitation of vulnerabilities harder*, and reducing the possible damage of the attack. To this end, we center our research around two software security techniques, diversification and obfuscation.

Code obfuscation is the process of scrambling the code and making it unintelligible (but still functional), in order to make reverse

* Corresponding author.
E-mail addresses: shohos@utu.fi (S. Hosseinzadeh), sjprau@utu.fi (S. Rauti), smrlau@utu.fi (S. Laurén), jmjmak@utu.fi (J.-M. Mäkelä), jjholv@utu.fi (J. Holvitie), sami.hyrynsalmi@tut.fi (S. Hyrynsalmi), ville.leppanen@utu.fi (V. Leppänen).

https://doi.org/10.1016/j.infsof.2018.07.007
Received 12 May 2017; Received in revised form 2 July 2018; Accepted 7 July 2018
Available online 10 July 2018

engineering more difficult [1]. The transformed code is functionally and semantically equivalent to the original code, but is more complicated and harder to comprehend [33]. With the help of code obfuscation, even if adversaries get access to source code, analysis of the code and finding the vulnerabilities will no longer be a simple task. This requires more time and energy and makes the reverse engineering of the code harder and more costly. Obfuscation does not guarantee that the program is not tampered/reverse engineered, but adds an additional level of defence by increasing the effort and cost for an attacker to learn the underlying functionality of the protected program. Various obfuscation techniques exist that obfuscate different parts of the code at different phases of software development process. For instance, using opaque predicates [75] is a common way of obfuscating the control flow of a program, at source code [109] or binary code level [247], at implementation [109] or compile-time [17].

Software diversification refers to changing the internal interfaces and structure of the software to generate unique diversified versions of it. The users receive unique instances of the software that all function the same, although differently diversified. In other words, diversification breaks the ”monoculturalism” and introduces ”multiculturalism” in the software deployment process.

Malware (malicious software) is any software that intends to run its code on user’s computer to disrupt the computer’s operation or manipulate the system towards the attacker’s desire [2]. To do this, it needs knowledge on how to interact with environment and access the resources. Software diversification alters the internal interfaces of the software and makes it challenging for malware to gain this knowledge. Thus, malware becomes incompatible with the environment and eventually becomes unable to take effective actions to harm the system. It should however be noted that, in order to maintain the access of legitimate applications to resources, we need to propagate the changes to trusted applications, i.e., they will be diversified as well to be compatible with inner layers.

Diversification does not attempt to eliminate the vulnerabilities of a software, but tries to avoid or at least make it toilsome for malware to exploit them and perform a successful attack. In a worst-case scenario, even if the malware succeeds in running its malicious code and attack a computer, this attack can only work on that particular computer. The designed attack model does not work on other computers, since their software are diversified differently with different diversification secrets. To take a large number of computers under control, different attack models should be designed specifically for each software instance, which makes it an expensive and arduous task for the attacker. On that account, diversification is considered as an outstanding approach for securing largely-distributed systems, and mitigating the risk of massive-scale attacks.

It is worthwhile mentioning that the terms obfuscation and diversification, sometimes, have been used interchangeably in the literature. In this paper, we make a clear distinction between these two concepts.

1.1. Method of the study

The method of study we chose in this research is Systematic Literature Review (SLR). A SLR is a means of research that identifies, evaluates and interprets all high quality studies related to a particular research question, or an area of interest [3]. This method of study was originally used in medical sciences [4], but later gained interest in other fields as well. A systematic review can improve a traditional review [4], in a way that the set of studies is not restricted to better-known and frequently-cited publications, and not biased towards the research area/interest of the researcher, as all studies in the field are captured. A systematic review, by classifying and mapping the scattered research studies, identifies research gaps and produce baselines for future research.

We conducted a SLR on studies that deal with the two techniques, obfuscation and diversification, with the aim of securing the code and software. There have been previously some other reviews [5,6,248]. However, they (a) cover a more limited number of studies (14, 69, and 10 papers respectively), (b) consider these two mechanisms from other perspectives than security, (c) focus on one of these two mechanisms, or d) discuss only one particular technique.

The surveys studying the obfuscation related studies include a review on control-flow obfuscation techniques [6], and a review on code obfuscation approaches [5]. These research works cover less than 15 studies and are published, respectively, in 2005 and 2006, which implies that the studies published after that are missing. Larsen et al. [248] authored a survey that reviews the state-of-the-art in automated software diversity with the aim of security and privacy. Another recent literature review on software diversification, surveyed by Baudry et al. [284], investigates diversification from five various perspectives aimed at different goals, including fault tolerance, security, testing, and reusability.

The main factors that differentiate our survey from the existing ones, are: (1) the systematic process for collecting the data, (2) a thorough list of covered studies on both obfuscation and diversification, (3) the focused scope of the study (security), and (4) classification and analysis of the collected studies.

1.2. Structure of the study

The remainder of this paper is structured as follows: Section 2 discusses the aim of our study, and specifies the research questions we have formulated and addressed in this research. Section 3 reports the process of search and selection of the relevant studies, and also the data extraction from these papers. Section 4 presents the results of the data collection and analysis of the results. In Section 5, we present the discussion. Limitations of the study, concluding remarks, and the future work come in Section 6.

2. Aims and research questions

We undertook a SLR of the papers reporting the use of obfuscation and diversification techniques in software security domain. Before starting the search, we determined the research questions, and formed the search strings. Our SLR addresses the following research questions:

- RQ1: What is the *aim* of obfuscation/diversification being used?
- RQ2: What is the *status* of this field of study? (E.g., outputs per annum, types of studies reported, collaboration of academia and industry)
- RQ3 In what *environments* the techniques are used/studied in order to boost the security (i.e., the programming language and execution environment the techniques are used for).
- RQ4: What *mechanisms* have been proposed/studied? (i.e., the obfuscation/diversification method used, (b) target of transformation, (c) level and stage, (d) cost and effectiveness of the approach.

3. Search and selection process

In order to carry out the research review systematically, we need to follow a protocol that defines the search strings and strategy, inclusion and exclusion criteria, and methods to extract data and synthesis the results. In this regard, we based our SLR on the research protocol suggested by Kitchenham et al. [7], and conducted our SLR in seven different phases. These phases are as follow: search and selection process (Phase I), inclusion and exclusion (Phase II to IV), snowballing (Phase V), data extraction (Phase VI), data analysis (VII). Fig. 1 illustrates the different phases in this process. The numbers on the arrows indicate the number of search results and included papers after each phase. In what follows, the details of the protocol developed for our SLR are presented.

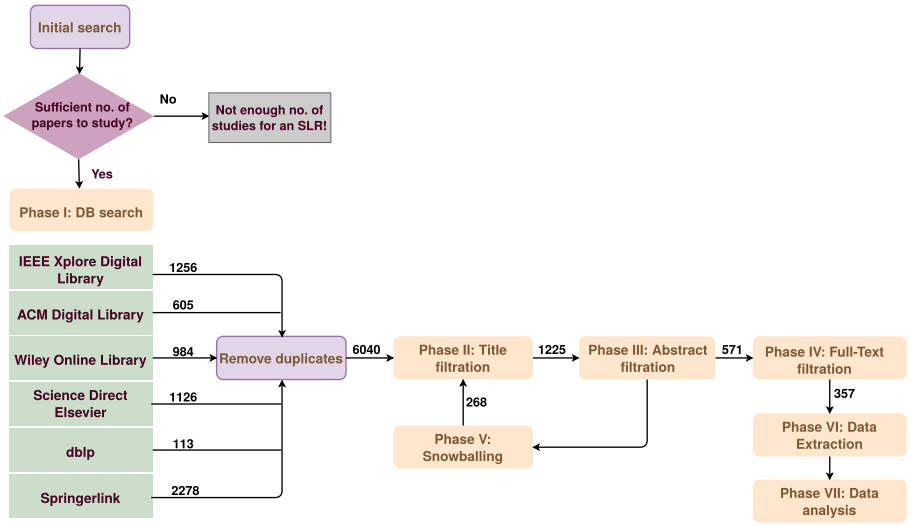


Fig. 1. The systematic search and selection process. On the left are the online databases and on the right are various inclusion and exclusion phases in the study. The number of articles left after each phase are shown on arrows.

3.1. Search

3.1.1. Initial search

Before starting the search process, we conducted an initial search to assure that there are sufficient numbers of articles available in the target field to study. In this stage, we found 48 articles discussing the improvement of software security using diversification/obfuscation techniques, which confirmed that this could be an appropriate topic to conduct a SLR on.

3.1.2. Manual search

For the manual search, Phase I, we selected a set of proper search strings, with which we assumed we would find the majority of the related articles. We also selected six of the largest digital databases, including IEEEXplore Digital Library, ACM Digital Library, Wiley online library, ScienceDirect, dblp, and SpringerLink. We limited our search to titles, abstracts and keywords of the articles to avoid false positive results of the full-text search. In some cases, search query was adapted according to requirements of the search engine. The following search command was used to retrieve studies from the databases:

```
(software OR code OR program) AND (diversification OR obfuscation
OR obfuscate OR obfuscator)
```

We undertook the manual search separately in the databases and combined the results in a large spreadsheet. After removing the duplicates, 6040 articles proceeded to Phase II for inclusion and exclusion (Section 3.2).

3.1.3. Automatic (citation-based) search

To complete the manual search, we performed an automatic search (backward snowballing), in Phase V. Backward snowballing is done by analyzing the reference lists of selected papers to find any missing related paper [7]. Therein, 268 papers were collected, for which we repeated the inclusion/exclusion process (Phase II to IV).

3.2. Selection of the studies

After collecting the papers in Phase I, we should include relevant and drop irrelevant papers. For that, we defined some inclusion/exclusion criteria, based on which we make decision (in Phase II-IV) whether to include/drop a paper. The followings are the inclusion criteria in our study:

- papers that are written in the English language;
- peer-reviewed papers (however, we did not exclude technical reports and books, since there exists some widely cited high quality technical reports in this domain, e.g., [13]);
- papers in the context of software production/development;
- papers related to software security;
- papers related to obfuscation/diversification; and
- obfuscation/diversification in the paper is used/discussed with the aim of improving/enhancing the security in software/code/program.

Considering that obfuscation and diversification techniques have been used in different domains for various purposes, we decided to narrow down our results. To this end, we focused our search on studies that are using obfuscation/diversification with the aim of software security and leave out the papers that were falling in our exclusion criteria:

- studying the possibility/impossibility of obfuscation;
- studying the use of obfuscation/diversification by malware, to hide their malicious code from scanners and malware analyzers;
- studying the techniques at a level other than software (e.g., hardware/network);
- proposing an approach that needs hardware assistance;
- studying obfuscation/diversification from cryptographic point of view;
- using the approaches to protect software watermark, birthmark and intellectual property rights; and
- unavailable studies, that we were not able to access in anyway.

Considering the defined criteria, we followed this process to select the relevant studies:

1. In Phase II, we screened the papers based on their titles. Each paper title was checked by four authors to determine whether it is relevant to our study or not, according to the defined inclusion/exclusion criteria.
2. In Phase III, two of the authors screened the papers based on their abstracts, and included the papers that were compatible with the inclusion criteria and dropped the papers that were not.
3. In Phase IV, the same process was repeated as Phase III, but based on the full text of the papers this time. There were several cases in which the full texts were not available in online databases. We tried

to contact the author(s) or find the text from other sources. If we were not successful finding the text in any way, we dropped the paper.

3.3. Data extraction

Each of the 357 selected papers was read through by two reviewers. The first reviewer extracted the data from the papers using a data extraction form, and the second reviewer checked the correctness of the extracted data. In case of any disagreement, the paper was discussed in a meeting with other authors, till reaching an agreement.

We divided the papers into two main categories, *Constructive* and *Empirical*, and defined different sets of questions to extract data form them. The papers that propose a new (implementable) obfuscation/diversification method, or apply/implement a technique fall into the category of constructive papers. The papers that evaluate/assess/experiment/discuss/review some (existing) obfuscation/diversification techniques fall into the category of empirical papers. There also exist papers that could be considered as both constructive and empirical. This class includes the papers that carry out an empirical study and at the same time conduct a constructive work.

For the category of *constructive papers* we extracted the following data, and presented the classification of the captured data in Section 4.1:

Aim: For what purpose is obfuscation/diversification used and what types of software security problems is solved (e.g., what type of attack is mitigated)?

Level: At what level is obfuscation/diversification applied (e.g., source code, binary level)?

Stage: At what stage of software production is obfuscation/diversification applied (e.g., compile-time, run-time)?

Target: What is the subject of obfuscation/diversification transformation (e.g., control flow)?

Mechanism: What type of obfuscation/diversification method is used/proposed?

Language: What language is the paper targeting?

Execution environment: What environment is the obfuscation/diversification techniques proposed for?

Overhead: What kind of overhead does the proposed obfuscation/diversification technique introduce?

Resiliency: How has the resiliency of the proposed approach been tested, and what results have been achieved?

For the category of *empirical papers*, we extracted the following data, and presented the classification of the captured data in Section 4.2:

Relevance: How is the paper related to obfuscation/diversification?

Outcome: What are the outcomes/findings/results of the study?

4. Results

As mentioned before, based on the method of the study used, we divided the selected studies into three main categories of (a) constructive, (b) empirical, and (c) constructive and empirical. Fig. 2 shows the distribution and the number of papers in each category. As is seen, the highest interest has been on constructive methods and obfuscation studies.

4.1. Constructive studies

By analyzing the data we captured from the data extraction phase, we answered the research questions defined in the beginning of our study.

4.1.1. RQ1: Status of the field of study

After the search and selection step, we extracted data from the 357 included studies. The studies come in six different types, including conference paper, journal article, workshop papers, book section, technical report, and doctoral theses. Also, there were 2 studies in other formats that did not fit into these categories. Table 1 shows different types of studies and the number of studies found in each type. The numbers indicate that the majority of the studies were published in conferences.

We analyzed the author affiliations for the included papers to associate the papers to their originating organizations and countries.

Fig. 3 captures the ten most associated countries for the considered set of studies. *United States* has by far the largest (c. 39, 5%) share, followed by *China* (c. 10,1%). However, as a continent, UK and Europe lead the statistics (c. 40,1%), with research divided mainly among Germany, Belgium, and Italy. The list also includes Japan and India – Asia as a whole contributed to one third (c. 32,2%) of the papers in the study. The research is relatively concentrated to a selected number of regions as the five and ten most affiliated countries count for circa 60,8% and 80,1% of all the affiliations.

Fig. 4 captures the ten most associated organizations for the considered set of studies. From this, we note that *Microsoft Corporation* (inclusive of Microsoft Research) is the only non-academic organization to be prolific in this area. Further, the ten most prolific organizations correspond to almost a third (c. 29, 1%) of the total affiliations for these studies. This is a notable portion from the affiliations, and arguably, indicates that majority of the research is concentrated to a rather small set of organizations. In Belgium, Finland, and New Zealand, the majority of research can be traced to a single organization.

It was of our interest to know the annual growth and decrease rates of the publications in this field of study. This can indicate the changes in interests and the significance of the field of study. An upward trend can be a sign of increasing interest to the field; while, a downward trend could state that the field is reaching a dead end. Fig. 5 illustrates the distribution of the selected studies in the SLR, between the years of 1993 to 2017. There is a relative fluctuation in the whole period, with an overall upward trend in the number of published studies, except for the slight decline in 2017. This implies that while the field has been fairly unpopular research subject, it has recently drawn fair attention among researchers. Between obfuscation and diversification, the former has almost always been a more popular technique – significantly so between 2000–2010, while diversification has grained in popularity since then.

We also examined the articles' publication forum types as a function of their publication years and the distribution is captured in Fig. 6. We note that through the queried year span, the dominant publication forum type is *conference*. However, the type selection gets more varied as we approach the present day, and as a publication forum, the *journal* type is almost on par with the conference in the year 2014. The observed increase in variety could be taken as evidence for the domain getting more mature: existence of more established research in the domain shows as increase in the number of journal articles and book chapters while the discovery of new sub-domains shows as an increasing number of workshop publications.

Fig. 7 displays, for the considered set of studies, the associated organizations' sector as a function of the publication year. Observations made here relate closely to the ones made for Fig. 4: while some publications are affiliated solely to industrial organizations (c. 2, 6% publications in the year 2015 and c. 5, 6% in total for the considered timespan) or to both industrial and academic bodies (c. 13, 2% in the year 2015 and c. 12, 6% in total), majority of the considered studies are made in an academic vacuum. While the distribution is understandable for theoretical research, it raises concerns regarding the applicability and correspondence of the research in this domain.

4.1.2. RQ2: Aim

In the reviewed literature, we identified a set of aims for which obfuscation and diversification were used for securing code and software and defeating known attacks, and hopefully unknown future attacks [238]. In Table 2, we summarize the generic aims that the related

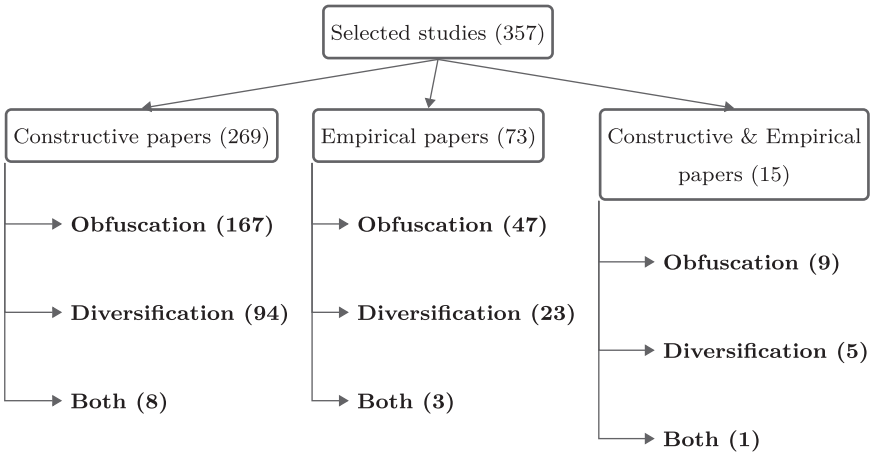


Fig. 2. Distribution of the studies.

Table 1
Types of studies.

Type	Diversification	Obfuscation	Both	Total
Conference paper	68	134	7	209
Journal article	29	51	2	82
Workshop paper	12	17	1	30
Book section	10	8	2	20
Technical report	3	8	0	11
Doctoral Thesis	0	3	0	3
Other	1	1	0	2
	122	223	12	357

studies were following.

In the process of reviewing the selected studies, we identified four broad categories that could encompass most of the presented literature. We acknowledge that these categories are not completely orthogonal, that is, there is some overlap between the different categories and a single piece of research could reasonably be classified as belonging to multiple categories. Still, being aware of the common aims or use cases associated with obfuscation and diversification research can be a valuable resource. With this classification, we try to answer the question what real-world problems are being solved by the use of diversification and obfuscation methods.

a) *Making reverse engineering of the program logic more difficult*: The most commonly stated aim of this research area was simply to make malicious reverse engineering of programs harder [113,165,171,277], i.e., making the act of debugging and disassembling of the software more complex to get the original source code [71,91,123,198,247]. By reducing the readability and understandability [47,110] of the software through these techniques, it becomes more resistant to unauthorized modification, i.e., becomes more tamper-proof [25]. Making understanding programs harder might be a desirable aim in order to protect proprietary algorithms or other intellectual property. Assembly code obfuscation [211], increasing complexity of dynamic analysis [240], preventing control-flow analysis [75], and introducing parallelism in order to obfuscate control-flows [239] are examples of research aiming to make programs harder to understand. Furthermore, obfuscation is an effective approach to counter both static [60,226,268] and dynamic analysis [122,126,240].

b) *Prevent widespread vulnerabilities*: Obfuscation and diversification techniques were also employed for their potential security benefits in preventing widespread vulnerabilities [81,262,268]. Exploits often depend on minute details about program internals. Introducing diversity into deployed applications can make it more challenging to construct exploits that reliably work against multiple targets. Diversification works by introducing variability in the software. Increased diversity makes the number of assumptions an adversary can make about the system smaller. Aside from diversification, obfuscation can also serve as a method of making software more secure. By making it more challenging for an attacker to understand the piece of software, obfuscation helps to increase the costs associated with exploit development. Examples of research specifically targeting security include randomization measures to defeat Return-Oriented Programming (ROP) attacks [216], randomized instruction set emulation [66], metamorphic code generation [230], and diversifying system call interface to defeat code injection attacks [159,233,282].

c) *Preventing unauthorized modification of software*: Research on tamper-resistance tries to find ways for making it more challenging for an adversary to produce derived version of programs [26,107,127].

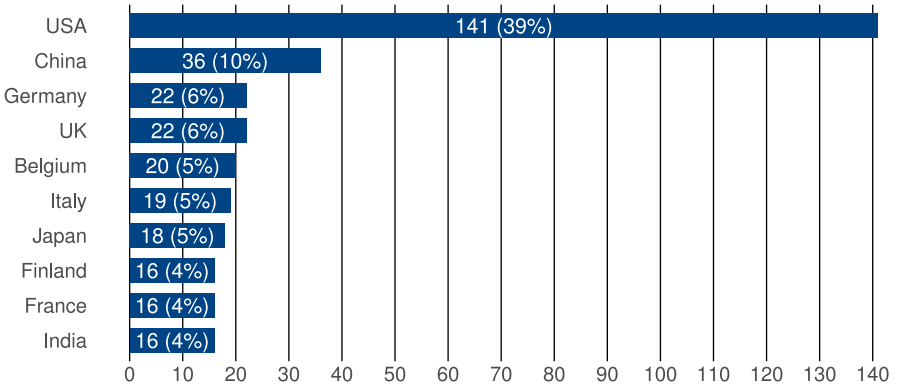


Fig. 3. Prolific countries: ten most associated countries in the considered studies (total number of country level affiliations $N = 420$).

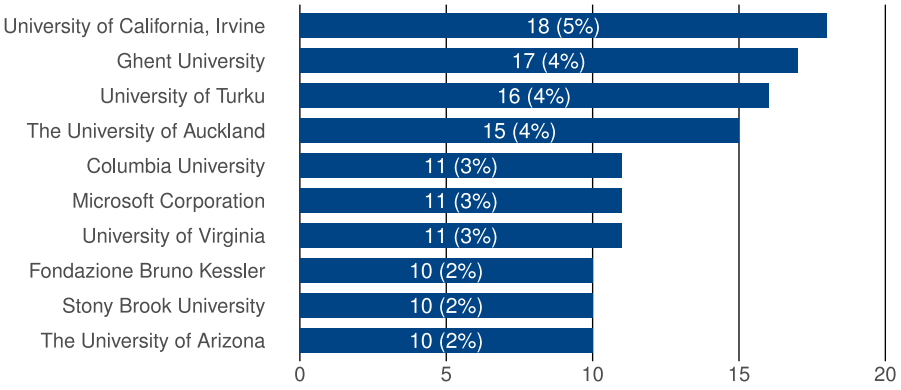


Fig. 4. Prolific organizations: ten most associated organizations for the considered set of studies (total number of organization level affiliations $N = 544$).

This might be desirable in order to preserve the intended operation of a program in an uncontrolled environment. For example, applications employing some form of digital rights management or computer games trying to prevent players from cheating might employ such techniques in order to make it harder to circumvent the protection mechanisms [30,259]. Techniques aiming for tamper-resistance often utilize methods for making understanding the program more difficult but they can also include methods for verifying program authenticity. Tamper-resistance was explicitly mentioned as one of the aims in the context of obfuscating Java bytecode [30], run-time randomization in order to slow down the adversary's locate-alter-test cycle [103] and obfuscation of sequential program control-flow [24]. Control flow obfuscation conceals the real control flows of the program and generates a fake control flow [145,175]. This makes it difficult for an analyzer to comprehend the logic of the program [245], also prevents spying and manipulating the control flow [75].

d) *Hiding data*: Aside from making programs more complex to analyze, obfuscation was also utilized for hiding static non-executable data within programs [99,231,281]. Hiding cryptographic keys and protecting intellectual property are few examples of scenarios were such measures are considered. Such techniques have been used to hide static integers [138,191] and obfuscate arrays by splitting them [97].

The results signify that the two techniques are used to mitigate the risk of a wide range of attacks, and in best case scenario hamper them. Table 3 presents the top attacks that were impeded with the help of obfuscation and diversification, such as code injection attacks [55,105,108,197], ROP attacks [195,215,260,263], buffer over-flow attacks [35,57,268], and Just-in-Time (JIT) spraying attacks [186,208,263]. From Table 3 we can deduce that not all the studies (209 papers) were explicitly discussing particular attacks that they aim to impede.

4.1.3. RQ3: Environment

For classifying the environments, two subcategories were chosen: a) *language* of the program being obfuscated/diversified and b) *execution environment*.

a) *Language*: The reviewed literature used a diverse set of over 20 different programming languages. Circa 36,8% of the languages were the topic of only one research and two thirds (63,1%) were mentioned at most thrice. Most research discussed one (c. 63,4%) or two (c. 10,6%) specific languages, with two systems programming (C/C++) or high level languages (Java & JavaScript) representing the vast majority of such pairs. A quarter (25,0%) of the research did not specify a single language or generalized the presented work for a class of languages. Only a minority of research [135,158,163,167,191,232,340] mentioned multiple languages or language classes.

A more descriptive view of the kinds of the languages was achieved by further classifying the research into four language categories representing hardware oriented, high level, scripting, and domain specific languages. The distribution of languages into these languages is as follows:

- Systems programming (N = 158): C (52), Assembly (29), C++ (21), Cobol (1)
- Managed (N = 81): Java (54), C# (3), Haskell (2), J# (1), Lisp (1), OCaml (1), VB (1)
- Scripting (N = 19): JavaScript (11), Python (3), Perl (2), PHP (1)
- Domain specific, DSL (N = 7): SQL (5), HTML (1)

The systems programming languages are compiled to native hardware without a run-time virtual machine and provide direct access to memory. Due to this low level direct hardware access, these languages benefit from obfuscation and diversification to protect this access. Some

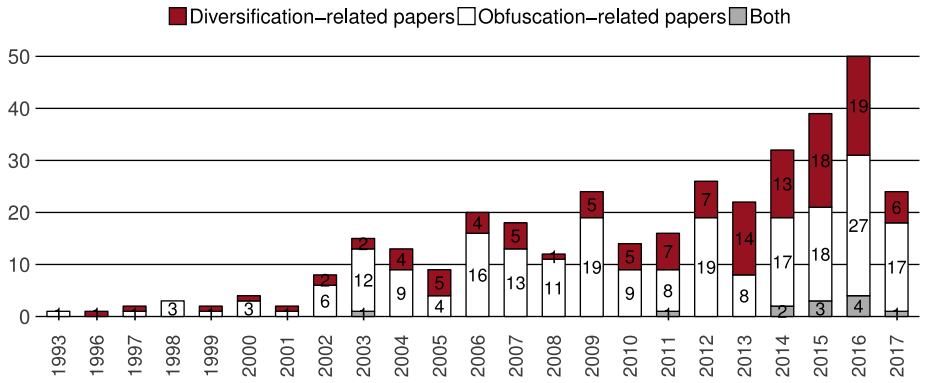


Fig. 5. Number of papers published yearly on the topic of security and privacy through obfuscation/diversification.

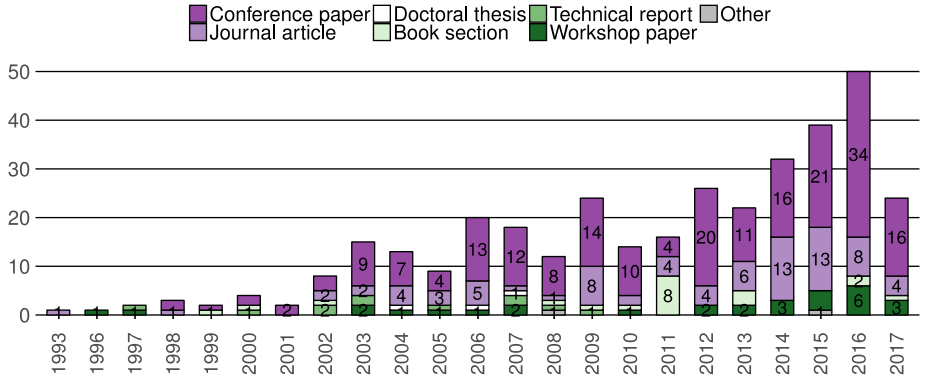


Fig. 6. Publication forum types for the considered set of studies as a function of the publication year.

examples of the applications of these languages in the research include operating systems and drivers, low-level libraries, server software, high performance computing, and embedded software. The managed languages typically require a virtual machine to provide a safer programming model for application programming. The most common problem for these languages is that the code is relatively easy to reverse engineer. The Java virtual machine is the most common platform in the selected research studies, but others, such as the Microsoft's Common Language Runtime (CLR), were also covered. A typical application of this class is mobile code, that is, code expected to run in an unknown environment. Finally, the scripting languages introduce new levels of insecurity since manipulating their code is even simpler. The DSLs have other issues, for example injection attacks or the need to protect intellectual property.

The following three figures present the language trends in the reviewed papers. First, Fig. 8 shows the popularity of various language categories based on our classification. The majority of the research has focused on systems programming, managed languages come as the second most popular category. Script languages are a bit more researched than DSLs.

Fig. 9 shows the overall distribution of language popularity in selected studies. A raw binary code (of native or virtual machine bytecode instructions) is the most popular "language" in this field of research. This is natural as most software is compiled to binary form for distribution. It represent the lowest level language and often requires disassembly to reconstruct the program structure for analysis. We distinguish assembly language as a separate form with its structured form intact for further analysis. Assembly is commonly used when obfuscation/diversification is used as a language agnostic compiler pass. The C and Java languages are other popular choices, followed by C++ and JavaScript.

Fig. 10 shows the trend over time for the five most used languages. The other languages are presented as the sixth group, as a reference. Like in the other figures, the research seems to be a bit more active in the 2000s and even more active in 2010s. Each of the top five languages appears to be almost equally represented each year.

b) *Execution Environment*: The environments in the reviewed literature can be classified in various ways as there are many interesting areas of focus. We have focused on two approaches in our review. First, the target environment of deployment (Table 4) plays a significant role when analyzing the applicability of a security mechanism. The majority of reviewed approaches are general enough to work in a multitude of environments. The most significant group of special environments were distributed and agent based systems with mobile code. As the code executes in a possibly remote, uncontrolled system, the need for protection is obvious - especially since the mobile agents often rely on bytecode that is relatively easy to reverse engineer. Virtualization and cloud computing can introduce similar kinds of problems if the host is owned by a third party, but virtualization is also used as a protection mechanism. Web services and servers offer an attack surface via the service layers, and mobile and desktop users are threatened by unreliable software. We distinguish between generic servers and cloud by denoting XaaS platforms for hosting third party services as the cloud. Embedded environment might use obfuscation or diversification for example to avoid the computational cost of encryption. Furthermore, most mobile devices are embedded platforms, but not all embedded platforms are mobile.

The second way to classify the reviewed literature is by the run-time environment (Table 4). This classification focuses on the abstraction level on the deployed software stack, with native code on the bottom and the virtual machine managed code on top, if both run-times are being used. Over a half of the research targets a native code environment. The more specific mechanisms are further discussed in the level

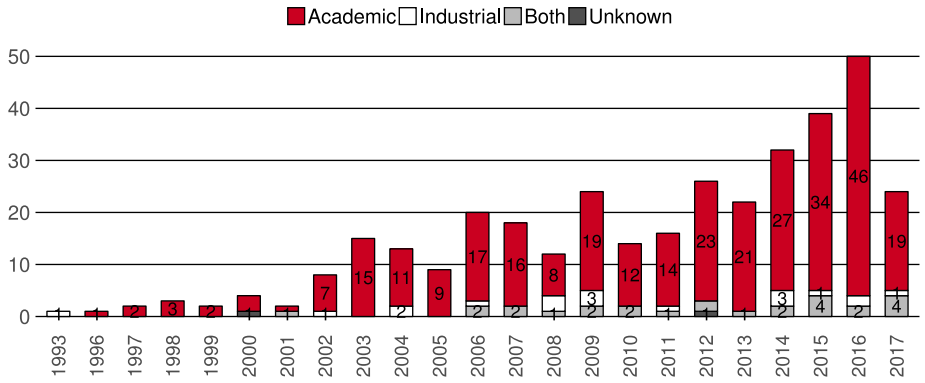


Fig. 7. Associated organizations' sector for the considered set of studies as a function of the publication year.

Table 2
Aims followed by using obfuscation and diversification techniques.

Aim	Via diversification (no. of papers)	Via obfuscation (no. of papers)
Making reverse engineering difficult	7	78
Generating diverse and unique versions of SW	34	3
Making the program hard to comprehend/read	1	31
Concealing a fragment of code and hiding some data inside the code	2	24
Preventing tampering of program code and illegal modification of software	4	22
Hiding the control flow of the program	1	24
Making static analysis difficult	1	20
Making dynamic analysis difficult	2	12
Mitigating the risk of malware	12	7
Protecting mobile agents against malicious host	0	6
Preventing large-scale attacks	10	2
Detecting anomalies/intrusions	4	0
No suitable aim discussed	50	

Table 3
Attacks mitigated by obfuscation and diversification techniques.

Attack mitigated	via diversification (no. of papers)	via obfuscation (no. of papers)
ROP attacks	24	1
Code injection attacks	15	2
Buffer overflow attack	6	2
JIT spraying attacks	2	2
Side channel attack	3	4
Attacks to web applications, e.g., cross-site scripting (XSS), SQL injection	4	1
Code reuse attacks	12	2
Browser-based attacks	2	3
Insider attacks	1	2
Protecting the software against piracy	0	6
Slicing attacks (a form of reverse engineering)	0	2
No attack mentioned	209	

(Section 4.1.4b) and stage (Section 4.1.4c) sections. Around fifth of the research focuses on managed environments such as Java virtual machines. Few papers target both environments, e.g., in the case of JIT compilers. Almost a third of the research claims to operate in all kinds of environments as a general purpose security mechanism.

4.1.4. RQ4: Mechanism

a) *Method*: In order to make diversified program instances, various transformation mechanisms are proposed in the literature. Each of these mechanisms are applied at different stages and levels of software development life-cycle (discussed in Section 4.1.4.b and Section 4.1.4.c). In this section, we classify the transformation techniques, based on the target of transformation. In other words, "what" is transformed and "how" the transformation is applied. Fig. 11 illustrates these techniques as a tree. On first level of the tree come the targets of transformations and on the lower levels the transformation techniques to obfuscate these targets. We base our classification on the taxonomy presented by Collberg et al. [13], which introduces *control obfuscation, data obfuscation, layout obfuscation*, and *preventive obfuscation* as different transformation targets. In the following we discuss each category.

- *Control flow obfuscation* aims at altering/obscuring the flow of a program to make it difficult for an attacker to successfully analyze and understand the code [1]. There exists a large body of research on control flow obfuscation techniques [259,261,266,334]. The most common technique to disturb the control flow is bogus insertion [11,12,22,31,63,73,104,268,362]. This technique works as inserting gray/dead/dummy code [351] that is never executed, fakes the control transfer [100], and/or introduces confusion for the analyzing tools [16,24,45,51,98,109,129,145,179,187] to attain the actual flow. Adding dummy blocks [122,160,169], dead statements [170], redundant operands [113], dummy instructions to camouflage the original instructions [38,242], new segments [247], dummy classes [84], dummy sequence using dead registers [47], and junk byte insertion to instruction stream [34,169], all fall into this class of transformation. Inserting additional NOP instructions [215,226,283] is another type of bogus insertion. NOPs are instructions that perform no operations but make it harder to predict where the pieces of code are placed in memory.
  Another widely used technique for disturbing the program's control flow is using opaque predicates [16,34,73,75,115,126,145,169,179,188,209,291,313,355]. These expressions are known to the obfuscator in advance, but not to the deobfuscator/attacker. A simple example of opaque expression is a Boolean expression that is

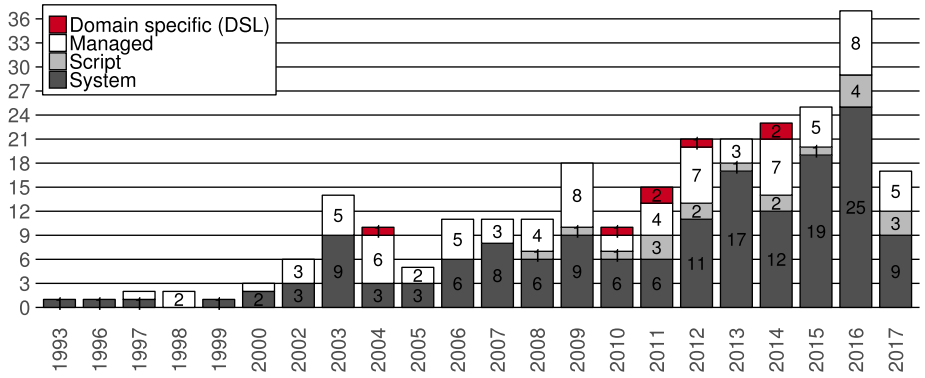


Fig. 8. Popularity of languages in the selected publications over time, grouped in language categories.

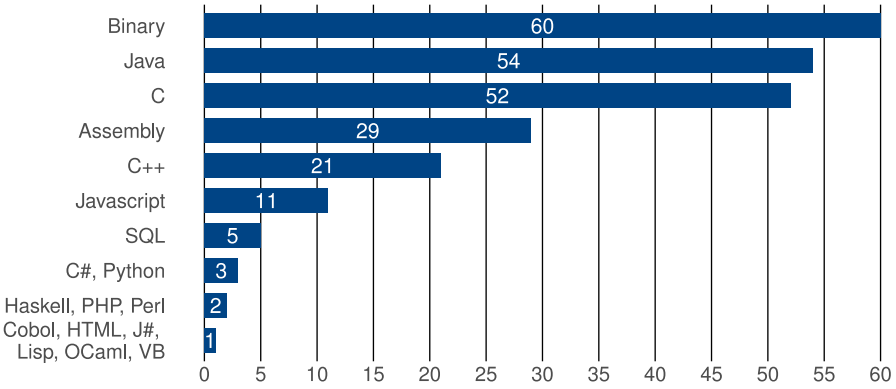


Fig. 9. List of languages in selected research, ordered by their popularity over time.

always evaluated as "true" or as "false", yet needs to be evaluated at execution time. This hardens the task of analyzing the control flow and enhances the cost of comprehending the program [16,75].

Transformation can be applied to loops [268] by loop unrolling [166,201,272], loop intersection [73,182], extending [16,104,113] and eliminating [109], and changing the loop conditions [330]. Transformation can also be applied at instruction level to camouflage the original flow of the application [271] through instruction reordering [103,114,166,245,268], instruction hiding [226], and instruction replacement with dummy/fake instructions [38,76,175,291,346], or instructions that raise a trap [47,103,186,247]. Self modification mechanisms [38,62,165,182,202] alter/replace instructions at run-time which could be used to introduce an additional layer of complexity while obfuscating the code [161].

Modifying the control of a program not only makes it difficult to analyze the actual program's flow, but also results in diverse binaries/executable. This can be achieved through reordering the instructions [103,114,166,245,268] and blocks [27,103,135,202,239,346], while the semantics and dependency relations are preserved. Code transformation [52,63,162,202,211,214,230] is another way of producing dissimilar binaries. As an example, by randomizing the software in a sensor network, the nodes receive diversified versions of the software [149].

Other forms of control flow obfuscation are polymorphism [44,84], branching functions [34,47,123,157,179,209,240], and transforming/faking/spoofing jump tables [34,242]. Inlining method [41,103] replaces the function call with the function body, so the function is eliminated and the primary structures are not disclosed. Cloning method [229,231] creates different versions of the function and tries to conceal the information about the function calls.

- *Data obfuscation* aims at obscuring data and concealing data structure of a program [207]. In the surveyed studies, various approaches have been used to this aim [259,279,288]. *First* is array obfuscation [29] that targets the structure (and the nature) of an array, trying to make it confusing to the reader. This can be done through splitting an array into smaller sub-arrays [97,112,130,171], or merging multiple arrays and making one larger array [130,171]. Other ways of array obfuscation are array folding [85,112,130,171], that increases the dimensions of an array, and conversely, array flattening [48,85,112,130,171], that decreases the dimensions of an array. *Second* is variable transformation to obscure/obfuscates variables [29,41,67,110,116,238,256]. Variables can be encoded [104], substituted with a piece of code [11], split into multiple variables [94,104,113], and vice versa, multiple variables can be merged together. *Third* is a more complex obfuscation technique, class transformation, which confuses the reader to comprehend the structure of a class [72]. This transformation includes class splitting into smaller sub-classes [36,41,128,177], merging/coalescing multiple

Table 4
Environments for the proposed obfuscation and diversification mechanisms.

Target environment context	Diversification	Obfuscation	Both
Cloud	5	3	2
Desktop	1	3	0
Distributed/agent based	18	9	2
Embedded	6	2	3
Mobile	13	5	1
Server/mainframe	4	12	0
Virtualization	7	7	1
Web	10	8	0
Runtime environment			
Any	54	21	6
Managed code	46	8	1
Native code	72	68	2
Both native & managed	4	1	0

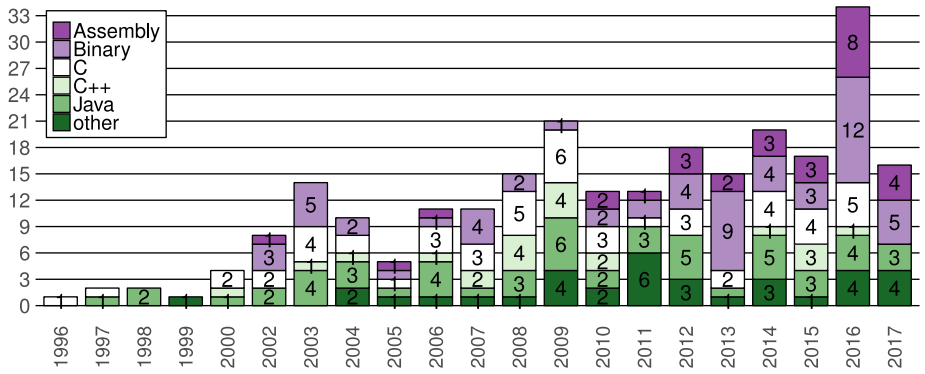


Fig. 10. Popularity of top five most used languages in the selected publications over time, the other languages are merged to the sixth group.

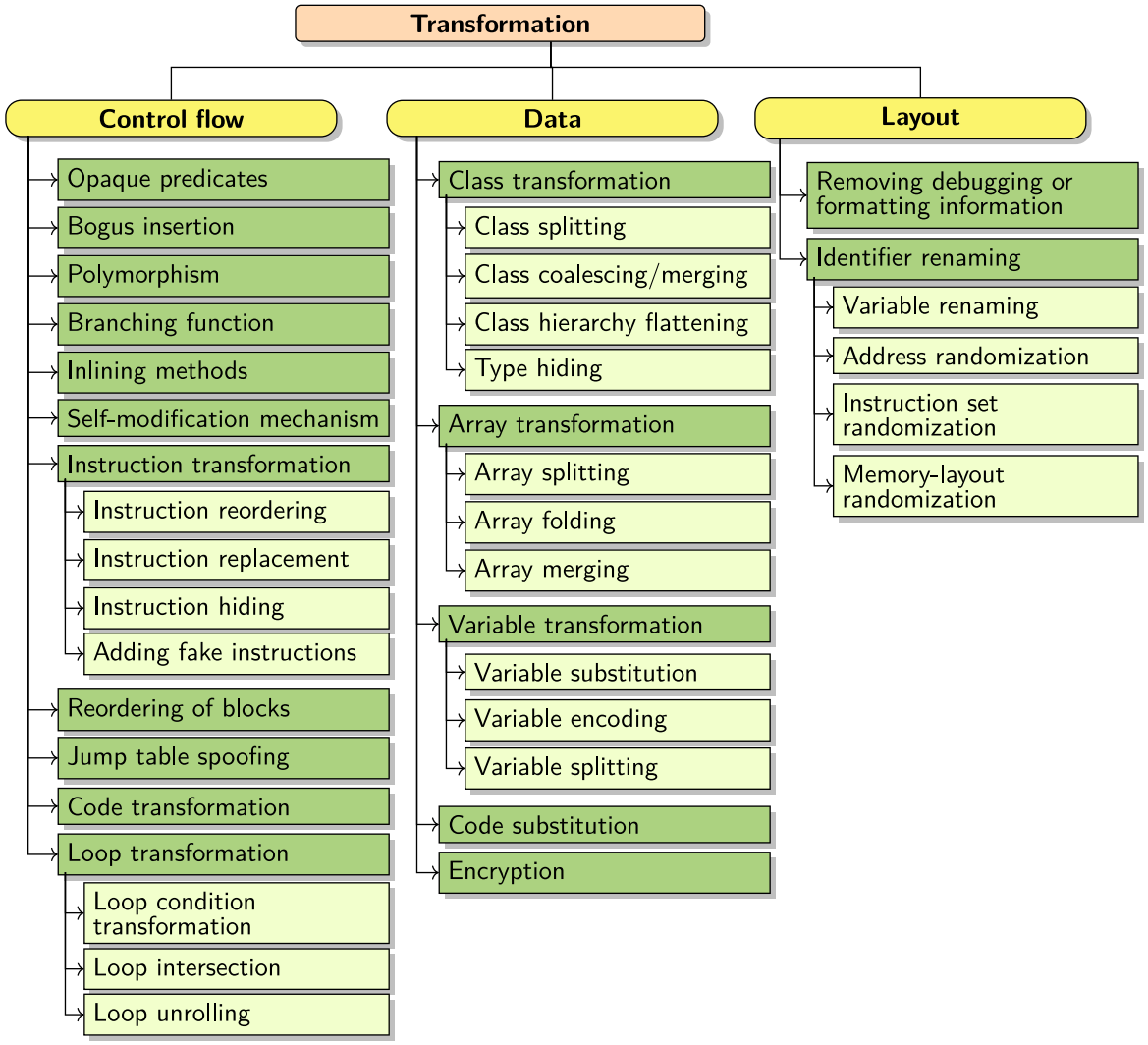


Fig. 11. Transformation mechanisms.

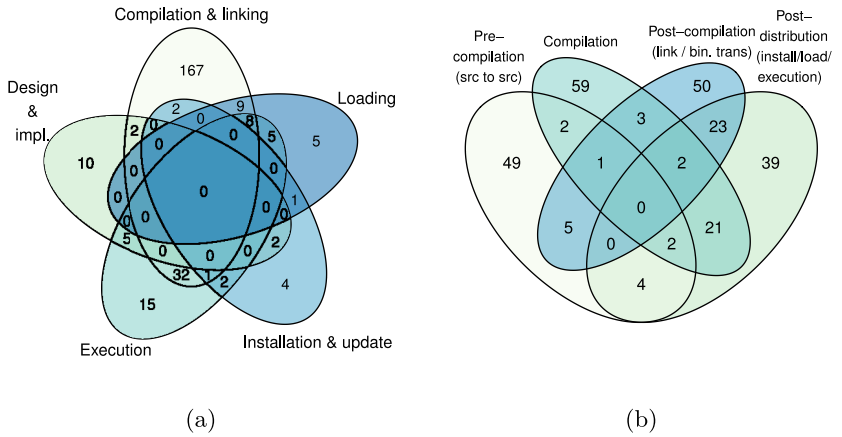
classes together [36,41,148,177,223], class hierarchy flattening [84,128,223] which removes type hierarchy from programs, and type hiding [36,72,177]. There exist other classes of techniques to obfuscate the data structure of the program, such as code substitution [145], and encryption [53,67,86,110,128,147,213,235].

- *Layout obfuscation* is a class of obfuscation techniques that targets the program's layout structure [13,336] through renaming the identifiers [45,51,98,101,110,117,125,163,187,212,213,233,320] and removing the comments, information about debugging, and source code formatting [113,170,201,223]. By reducing the amount of information for the human reader, the reverse engineering becomes harder. Layout transformations are considered as one-way approaches, as when the information is gone there is no way to recover the original formatting. Instruction Set Randomization (ISR) [55,66,105,140,154,158,167,186,192], Address Randomization [35,39,46,57,105,106,192,193,215,283], and Layout Randomization [41,52,88,113,146,149,160,178,193], Address Space Layout Randomization (ASLR) [263,265,308,337] can also be seen as identifier renaming techniques.

b) *Level*: We identified several phases in the software development, deployment, and execution as levels of obfuscation. In the reviewed research (Table 5), most techniques apply to development time (n = 282), runtime (n = 95), or both (n = 58). The development time techniques mostly apply to human readable source code (high level language & assembly), but obfuscation and diversification tools manipulating the generated binary formats (bytecode, native code, intermediate representation) are equally common. The application program itself provides the main platform for applying various mechanisms. At runtime, the techniques either target the source code (scripting languages), intermediate formats (e.g. JIT compilation), or the execution environments. Modified runtime systems are process level techniques for both managed (e.g. CLR & Java virtual machine) and native code,

Table 5
Level of obfuscation and diversification at development time / runtime.

Level	Development	Runtime
Application design	11	–
Assembly source code	12	–
Bytecode	40	–
Executable	76	–
High level language source code	104	7
Intermediate representation format	39	5
Managed code	–	3
Native code	–	43
Hardware	–	3
Operating system	–	18
Virtualization	–	16
Total no. of papers (impl & runtime effects)	58	
Total no. of papers	282	95



(a) (b)

Fig. 12. a) Various stages in SW life-cycle that obfuscation/diversification are applied on, b) Dissection of various compile-time stages in conjunction with all post-distribution phases.

but operating systems, operating system / machine virtualization, and hardware level modifications are also presented.

In terms of obfuscation and diversification techniques, operating with *source code* that is not yet compiled is relatively effortless. Many of the reviewed techniques work purely on the lexical and syntactic levels and the parsing technology is mature, ranging from simple pre-processors to frameworks with compiler-like abilities. It is also possible to manipulate many high-level structures (classes, data structures) that are not available in machine code form [36]. In interpreted languages (e.g. JavaScript), the source code obfuscation is the only option [110], which also explains why some of the source code obfuscations are deferred to run-time. Collberg et al. have extensively described techniques available for source code obfuscation in [13,16]. Some of the mechanisms extend the range of obfuscations to semantically richer forms, the intermediate formats available during the compilation. Abstract syntax trees [133,207] are used by syntax oriented techniques while semantically richer intermediate formats provide access to e.g. control flow analysis. These mechanisms are provided for instance as compiler plugins.

The motive to obfuscate the source code is usually preventing the adversary from easily understanding and altering the code even if he or she has managed to reverse engineer it. Source code obfuscation might not ultimately prevent a dedicated attacker from understanding software, but it will significantly raise the bar of complexity and decrease the probability of a successful attack [49]. Source code obfuscation is often used for intellectual property protection [104,113]. Worth noting is that source code obfuscation is usually also reflected to the bytecode or binary code after compilation.

In managed environments, *bytecode* techniques have received lots of attention. For example, in Java, it is not that hard to reverse the compiled bytecode back to source code. This reverse-engineering can be performed via automatic tools [50,51]. Naturally, this poses problems for the confidentiality of source code and has elicited lots of research on bytecode obfuscation. Several approaches such as [30,45,177,182,223] have been proposed to prevent adversaries from understanding, reverse-engineering or cracking the bytecode. One major advantage of bytecode obfuscation (along with other binary code obfuscation techniques discussed next) is that source code is not needed in the process. This is quite often the case with closed source, third party software.

Reverse engineering and the manipulation of security measures are also issues with native code executables, but the native instruction sets are inherently harder to analyze due to more complex instruction sets and the lower level of abstraction. A large set of obfuscation and diversification techniques are applied to symbolic assembly code (with relocation information etc. intact) or disassembled final binaries [259,276]. This is often done in order to make reverse engineering considerably harder [171,247] or to prevent disassembling the program from binaries [175,240]. Low level obfuscation usually involves using control flow obfuscation transformations changing the sequence of instructions [123,245]. In general, increasing the entropy of the low level code also makes it harder for a piece of malware to modify the code or inject its own malicious payload [11,233]. One technique related with low level obfuscation is ISR [42,66]. An execution environment unique to the running process is created so that the attacker does not know the ”language” used and therefore, cannot ”talk” to the machine. A new instruction set is created for each process executing within a system.

c) *Stage*: Although modern software development is iterative, we observe the software life cycle as a linear sequence of stages: (a) development, (b) distribution and deployment, and (c) execution. This model captures the fact that each stage is characterized by a different set of obfuscation and diversification techniques and tools. The development stage is further split into design, implementation, compilation and linking phases. When analyzing the types of tools used to manipulate the application’s code, the compilation can be further refined into pre- (e.g. source to source transformations and code generators) and post-compilation (e.g. link-time code transformation) phases. The software deployment includes installation and updates [248]. Application loading occurs in conjunction with execution and thus is included in this stage. The surveyed studies discussed and applied obfuscation and diversification techniques during all these stages.

The Venn diagram in Fig. 12a illustrates all the observed stages and their overlap in five main groups, from design to application execution. These groups reflect the different stakeholders and roles in the software’s life cycle. We identified 16 different types of use of stages, with 201, 60, and 9 studies operating on one to three stages, respectively. None of the studies suggested taking part in four or more groups of stages. A majority of research involves compilation and linking. Execution time techniques form another large group. A small number of research is associated with either of these approaches and some other stage (n = 29) or is applied outside these stages (n = 22).

In Fig. 12a, the first group contains design and implementation phases. Mechanisms applied at this stage are involved in software development effort. Data obfuscation [118,137,162], control flow obfuscation [109,169] and, in general, source code obfuscation [63,99,118,188,225] are the most common approaches that target the code at *implementation level*.

The mechanisms in the next group, compilation and linking, can be applied to the deliverables of iterations for in-house software or to premade software, available either as source code, in intermediate forms, or as executable binaries that can be analyzed or reverse-engineered. This group is further dissected in Fig. 12b as the majority of reviewed

literature forms a cluster in this stage. The third stage, installation and update, includes the task of local deployment of software and updates. The next stage, loading, covers the process of loading the executable to memory (e.g. from a network stream or disk) and dynamic linking. Finally, execution stage includes all sorts of mechanisms that activate during application's execution. Code obfuscation and software diversification can also be applied at *execution time*. Dynamic software mutation [79] is a repeated transformation of the program during its execution. It makes a region of memory occupied by various code sequences during execution. Identifier renaming [163], ASLR [57,265], camouflaging the instructions by overwriting them with fake instructions [76], and randomizing the location of critical data elements in memory [140] are other examples of execution-time diversification.

Fig. 12b focuses on the mechanisms applied on various stages of the compilation (pre-, post-, and during compilation). In the figure, all the remaining stages after the compilation techniques have been combined as a single post-distribution stage. The reviewed research is distributed quite evenly between the different compilation stages. Second large class of mechanisms is to use compilation or post-compilation in conjunction with the execution time techniques.

Diversification at the *compile-time* makes the process fairly automatic by eliminating the need to change the program's source code [248]. M. Franz [150] has proposed a practical approach for generating diverse software at compiler-level. This approach is based on an app-store that contains a multi-compiler, which works as a diversifying engine that generates unique binaries with identical functionality. In [207] they have developed a compiler plugin to generate diverse operating system kernels, through memory layout randomization. As we mentioned before, in the literature, there are several works that study the pre-compile time and post-compile diversification. Control flow transformation in the source code [43] is an example for the former, and class transformation in Java bytecode [177] an example for the latter.

d) *Cost*: Despite of the security obfuscation and diversification bring, they introduce cost and overhead to the system, like any other security measure. In fact, the higher level of obfuscation/diversification, the more penalty is forced to the system. Therefore, based on the need of the system, it is decided how much the program needs to be obfuscated/diversified. In the studied works overhead mainly was reported as a) increase in the program size [50,84] (e.g., number of instructions [34], memory size, code size [240,256,264], binary patch size [140], byte code size), b) increase in program performance [261,263,266,285,290] (e.g., compile time [175], process time, execution time [260,268], CPU overhead [119], higher memory usage [119,273], and c) latency and throughput [210] (in load time or run time). It is worth mentioning that among the diversification mechanisms, some introduce more cost and some less. For instance, changing variable names, function names, and system call numbers often introduces no additional costs.

e) *Effectiveness*: In the studied works, the effectiveness of the proposed approaches were mainly measured through the following metrics:
   - *Potency* determines to what degree a human reader is confused, as a result of the applied security measure [13]. Measuring the potency can be done by comparing the obfuscated/diversified version of the software with the original version and presenting the similarity rate [257,290]. In [131] clone detection is used to analyze the similarity of the obfuscated code with the original one, and the code dissimilarity is the metric for representing the potency of the approach. Another way to measure the potency of an obfuscation mechanism is to evaluate how much harder it has become for a human reader to comprehend the obfuscated code, comparing to the original code. For instance, the obfuscation mechanism in [27] has been tested empirically with a group of students, programmers and crackers and illustrated that only a few crackers were able to deobfuscate the obfuscated code. In [272] the effectiveness of the proposed approach is measured by static and dynamic analysis of the obfuscated code.
   - *Resiliency* determines how well the obfuscated/diversified program resists automatic decompilers/disassemblers/deobfuscators [13]. Analyzing the reverse engineering effort demonstrates how the proposed technique is effective against disassembly tools (e.g., through presenting confusion factor, and disassembly errors). For instance, in [211,240] the strength of the obfuscation mechanism has been evaluated against IDA PRO automated deobfuscators [8], and demonstrate that obfuscated code increases the effort for an attacker, by making it harder to reconstruct the original code. Similarly, Linn et al. [34] have used three state-of-the-art disassembly tools, and demonstrated the effectiveness of their approach through confusion factors, disassembly errors, and incorrectly disassembled code, that they gained by disassembling the obfuscated code.
   - *Attack Resistance* determines how much harder it has become to break the obfuscated code. It can be done by running the obfuscated/diversified software against different attacks and analyzing the outcome [260,263,268,285]. As an example, the obfuscated kernel in [207] is tested against four kernel rootkits, and it is shown that they all were disabled. RandSys prototype [105] implemented for Linux and Windows has been tested against two zero-day exploits (code-spraying attacks), and 60 existing code injection attacks. It was shown that the approach is successful in thwarting them. In [268] they run the program against various types of attacks (e.g., code injection, memory corruption, code reuse, tampering and reverse engineering attacks) and measure the resistance.

4.2. Empirical studies

As mentioned before, in the set of studies collected, 68 of them were studying the obfuscation/diversification techniques empirically. These empirical studies come in the form of discussion, experiment, evaluation, comparison, optimization, survey, and presenting a classification. The following categories illustrate how these studies were related to obfuscation and diversification:

- survey of related works on obfuscation and diversification as software protection techniques [5,6,37,64,180,248,284]; Baudry and Monperrus [284] survey the related works on design and data diversity which consider fault tolerance and cybersecurity. They also study randomization at various system levels.
- overview/classification of existing obfuscation/diversification techniques [59,78,132,324];
- studying the obfuscating transformations that are (more) resilient to slicing attacks [92,96,329];
- comparing different obfuscation mechanisms [87,95,190];
- discussion on a particular obfuscation mechanism [19,78,132]; In [132], obfuscation is being discussed as a way to make understanding the software more difficult. In [19], identifier renaming is discussed as an obfuscation mechanism to protect Java applications. By making the classes harder to decode, the act of unauthorized decompilation becomes difficult. In [78], the authors overview the existing obfuscators and obfuscation mechanisms, and also illustrate the possibility of achieving binary code obfuscation through source code transformation.
- studying/evaluating the effectiveness of an obfuscation/diversification approach (e.g., identifier renaming and opaque predicates) against human attackers [54,64,74,93,176,183,246,266,269,278,289,295]; In [68] the authors qualitatively measure the capabilities

and performance of two commercial obfuscators for three different sorting algorithms. In [93], the effectiveness of decompilers and obfuscators are quantified through a set of metrics. It is done by comparing the original Java source code with the decompiled and obfuscated code respectively. The metrics will then measure whether the decompiler produces valid source code, and whether the obfuscator produces garbled code. [176] measures the effects of different obfuscation techniques on Java code in terms of complexity. In [278] several different metrics are suggested for measuring the incomprehensibility of the obfuscated code.

- optimizing and reducing the overhead of software diversity [83,202,237,255];
- experimenting and evaluating the potency of an obfuscation technique; The strength and incomprehensibility of the obfuscated programs can be evaluated by measuring the performance of human analyzers in analyzing the obfuscated code (to what degree a human reader is confused) [111,121,145,228,234,354].
- studying the effectiveness of software diversity [32,65,136,237,274,287,292,360]; For instance, to evaluate the effect of diversity, several different computer attacks are tested against the diversified programs [32]. In [274], automatic software diversity is discussed as a means for securing the software. The authors investigate the types of exploitation it can mitigate, the different levels of software life-cycle the diversification can be applied at, and the possible targets of diversification.

5. Discussion

The idea of protecting software through generated diversity and obfuscated code originated in early1990′s and gained more attention in the past decade. The rationale behind these techniques is to increase the cost and effort for a successful attack. This study, by surveying the literature about the use of these two techniques for securing software, elucidates several points.

First, these methods have been used in various ways with different aims, such as protecting software from malicious reverse engineering and tampering, hiding some data and protecting watermark information, preventing the wide spread of vulnerabilities and infections, mitigating the risk of massive-scale attacks, and impeding targeted attacks. In a previous study, we have studied the aims and environments that these two techniques have been applied to [324].

Second, the field has grown in many directions, and new areas have emerged. Moving Target Defence (MTD) [172] is an example of newly born defence mechanisms. MTD randomizes the system components, and presents a continuously changing attack surface, which shortens the time frame available for attacker.

Third, studying all the related works sheds light on the research gaps, and the potential research directions. We discuss these research gaps in Section 5.2.

Fourth, There are also some challenges associated with practical diversification and obfuscation that require further study. In Section 5.1 we discuss some of these challenges.

5.1. Challenges

One of the challenge of applying diversification is the fact that it has to be propagated to every parts of the system that makes use of the diversified interface. For example, diversified system call numbers in the operating system kernel have to be propagated to libraries that employ system calls. It is not straightforward to automatically find all the dependencies.

Software updates are also a challenge for diversified systems, because each update has to be individually diversified to be compatible with the uniquely diversified interfaces it uses. Then again, each patch is also an opportunity to re-diversify some parts of the system.

Using diversification and obfuscation is always a race against time in the sense that the adversary will ultimately be able to guess the correct diversification key or deobfuscate the code. Dynamically changing obfuscation/diversification is a good defence, but raises complexity and performance costs of the approach.

5.2. Research gaps

As mentioned before, the main goal of conducting a systematic literature review is to collect and analyze the studies related to a field of research, in order to pinpoint the research gaps in that field.

One of the gaps we have found in this field of study is the lack of a standard metric in this field for reporting the overhead and also the effectiveness of the proposed approaches. The terms, *potency* and *resiliency*, introduced by Collberg et al. [16] discuss how much more complex the code has become in the presence of the obfuscation method; however, we believe that a standard metric to present a numeric degree is missing. Moreover, as also discussed in [276], the majority of the existing research are constructive papers and there is need for more empirical research, such as measuring the efficiency of diversification and presenting results on performance and space requirements of the obfuscation techniques. However, these concerns have not fully addressed by the research community.

Another research gap that we noted was that there are still many environments obfuscation and diversification have not been applied to yet, even though this would potentially be beneficial. In this section, we discuss these environments and present some ideas on how to utilize the two techniques as a complementary measure to the security measures these environments already have.

One of these execution environments is cloud computing and virtual environments, in general. Lately, virtual technologies have become very dominant as many enterprises and service providers are shifting towards the cloud and to deliver their services through it. Thus, we believe that due to the significance of these environments, there are needs for proactive approaches for securing their software. Obfuscation and diversification could become helpful in this manner. Our recent survey on the use of these techniques for securing cloud computing environment [293] clearly shows that there have been very limited number of studies on this domain and there is still room for more studies in this area. Also, the related works do not address the propagation issues, especially regarding propagation to higher level interfaces/APIs. One possible solution that we propose here is to diversify the internal interfaces of cloud and virtualization systems, for instance, diversifying the machine language of the virtual machines.

Therewith, container-based virtualization is drawing more attention recently, because of the advantages that it has compared to traditional hypervisor-based virtualization (such as higher efficiency in CPU, memory, and storage). We believe that this environment can also benefit from diversification techniques. However, most notably there is hardly any research related to diversification of hypervisors or containers. One potential idea is to diversify interfaces and APIs of containers, so each container would have a diverse execution environment.

IoT network is another type of environment that is becoming prevalent more and more these days. Protecting these networks is also crucial not only at network level but also at (device) application level. In the reviewed literature, obfuscation and diversification have been used very little as a security measure for this purpose. Therefore, this is another research direction to consider. We have proposed applying these two techniques on the operating systems of the IoT devices and also on the communication protocols used among them [275,322]. Then, in another study we applied diversification on Thingsee and

Raspbian operating systems [305] (the limitations of this study is discussed in Section 5.3).

These ideas could be also extended to fog computing as well. Looking at the architecture of fog computing, nodes (i.e., sensors, devices and data collectors) come at the edge of the network. Applying diversification at the nodes stops the malicious activity right at the edge before it goes up to the fog.

As mentioned earlier, diversification techniques could be applied as a complementary security approach, to the measures environments already have. We are studying applying diversification in trusted environments. More specifically, we consider diversifying the containers and storing the diversification secret inside TPM. Another idea is diversifying enclaves in a trusted execution environment such as Intel SGX.

Applying obfuscation to mobile platforms is also a topic that has received some attention recently but still requires more research. For example, Android applications, distributed in Dalvik bytecode form, are vulnerable to reverse engineering. There has been some positive development in this front, such as the ProGuard obfuscation tool being packed along with Google's Android Studio development environment and some recent publication on obfuscation in the context of mobile applications [35]. Most current studies on obfuscation in mobile environments, however, only highlight obfuscation techniques Android malware uses to evade detection. Therefore, with applications run on mobile platforms getting more prevalent, new obfuscation and diversification techniques to increase software security and research on their resilience are still needed.

To sum up, by systematically studying the two software security techniques, obfuscation and diversification, and finding the research gaps and research opportunities, we believe that both research community and industry should study and apply these techniques. There has already been some development in this direction. As we saw in Section 4.1.1, research on these techniques has drawn fair attention in software security community. This trend is also shown by several patent applications filed during the last ten years. For example, there are patents on ISR [9] and interface diversification [10]. There have also been some initiatives by Microsoft research center [119]. Still, we would like to encourage the practical application of these techniques even more – especially by industry.

5.3. Limitations of the study

As discussed earlier, a systematic literature review has a number of advantages compared to the traditional literature reviews, such as transparency, larger breadth of included studies, and reduction of research bias [4]. Nevertheless, this method has several different practical challenges and limitations as well, that we experienced in this work that made it difficult to put it into practice. First, it was very time-consuming and arduous work, due to the high number of included studies, in all the process of search, selection, data extraction, and also synthesis of the data and classification of them. Second, we selected the set of search strings manually. Thus, there is a concern that not all the materials in the field are captured. Third, the inclusion or exclusion of the studies might be biased based on the researcher's knowledge. Moreover, because of high number of authors, sometimes, each author had slightly different interpretations of the research questions. To solve the inconsistency in the screening process, we solved the disagreement cases in the meetings with the presence of all authors. Fourth, in some cases the search flow was dictated by limitations of some of the databases. As an example, in SpringerLink digital library we had to make the search in the full-text of the studies, which resulted in a huge number false positives. For other databases, we had limited the search to the title, abstract, and keywords of the studies.

In addition, like any other security measure, despite of the security benefits that the two studied techniques have, they also have some limitations such as increase in the size of the obfuscated code, and consequently increase in the execution time. In diversification method propagation of the diversification throughout the whole system from the lower levels to the upper most layers, is still a challenge and needs improvement.

Moreover, not all the diversification techniques are well suited for the tiny resource constrained devices. In other words, the diversification method should be chosen with consideration. For example, embedded systems may not always have Memory Management Units (MMU), which makes applying ASLR less worthwhile. We can perform device-specific diversification with the layout, but the embedded system requires certain offsets to be static, which means since there is no MMU we cannot hide these offsets with ASLR, which could mean all the diversification was for nothing since the exploiting code can possibly crawl through the known offsets to the offsets it needs to function. Diversification can expand the size of the system, which means the space constraints of such tiny systems come at us faster. In our previous experiments, for example, applying layout shuffling and symbol diversification required some extra space [239]. In Thingsee OS the size of binaries expanded, but not exceedingly.

6. Conclusion and future work

Obfuscation and diversification are promising software security techniques that protect computers from harmful malware. The idea of these two techniques is not to remove the security vulnerabilities, but to make it challenging for the attacker to exploit them and perform a successful attack.

In this study we reviewed the studies that were applying/studying obfuscation and diversification for improving software security. For this purpose, we systematically collected and reviewed the related studies in this field, i.e., 357 articles (See Appendix A), published between 1993 and 2017. We reported the result of study in form of analysis and classification of the captured data, and also we managed to answer the formulated research questions. We found out that these two techniques have been utilized for various aims and mitigation of various types of attacks, they can be applied at different parts of the system and at different phases of software development life-cycle. Moreover, in the literature, there exists many different techniques to obfuscate/diversify the program that each present different levels of protection, but also overhead, which depending on the need of the program could be chosen.

Moreover, by studying the existing works we pinpointed the research gaps. We concluded that the major part of the existing research works were focusing on obfuscation, and there is still room for studies on software diversification. We discussed that there exist many different environments that could still benefit from these techniques and need more focus of research, such as virtual environments, IoT, fog computing, and trusted computing. For the discussed environments, we also presented some potential ideas.

As future works, we will apply diversification on the interfaces of containers, also diversify the codes inside enclaves in trusted execution environments.

Acknowledgment

The authors gratefully acknowledge Tekes – the Finnish Funding Agency for Innovation [grant number 3772/31/2014], DIMECC Oy, and the Cyber Trust research program for their support.

Appendix A. Selected studies

The following is the list of 357 studies reviewed in this SLR, sorted based on the publication year:

1993	[11]				
1996	[12]				
1997	[13]	[14]			
1998	[15]	[16]	[17]		
1999	[18]	[19]			
2000	[20]	[21]	[22]	[23]	
2001	[24]	[25]			
2002	[26]	[27]	[28]	[29]	[30]
	[31]	[32]	[33]		
2003	[34]	[35]	[36]	[37]	[38]
	[39]	[40]	[41]	[42]	[43]
	[44]	[45]	[46]	[47]	[48]
2004	[49]	[50]	[51]	[52]	[53]
	[54]	[55]	[56]	[57]	[58]
	[59]	[60]	[61]		
2005	[62]	[63]	[64]	[65]	[66]
	[67]	[68]	[69]	[70]	
2006	[71]	[72]	[73]	[74]	[75]
	[76]	[77]	[78]	[79]	[80]
	[81]	[82]	[83]	[84]	[85]
	[86]	[87]	[88]	[89]	[90]
2007	[91]	[92]	[93]	[94]	[95]
	[96]	[97]	[98]	[99]	[100]
	[101]	[102]	[103]	[104]	[105]
	[106]	[107]	[108]		
2008	[109]	[110]	[111]	[112]	[113]
	[114]	[115]	[116]	[117]	[118]
	[119]	[120]			
2009	[121]	[122]	[123]	[124]	[125]
	[126]	[127]	[128]	[129]	[130]
	[131]	[132]	[133]	[134]	[135]
	[136]	[137]	[138]	[139]	[140]
	[141]	[142]	[143]	[144]	
2010	[145]	[146]	[147]	[148]	[149]
	[150]	[151]	[152]	[153]	[154]
	[155]	[156]	[157]	[158]	
2011	[159]	[160]	[161]	[162]	[163]
	[164]	[165]	[166]	[167]	[168]
	[169]	[170]	[171]	[172]	[173]
	[174]				
2012	[175]	[176]	[177]	[178]	[179]
	[180]	[181]	[182]	[183]	[184]
	[185]	[186]	[187]	[188]	[189]
	[190]	[191]	[192]	[193]	[194]
	[195]	[196]	[197]	[198]	[199]
	[200]				
2013	[201]	[202]	[203]	[204]	[205]
	[206]	[207]	[208]	[209]	[210]
	[211]	[212]	[213]	[214]	[215]
	[216]	[217]	[218]	[219]	[220]
	[221]	[222]			
2014	[223]	[224]	[225]	[226]	[227]
	[228]	[229]	[230]	[231]	[232]
	[233]	[234]	[235]	[236]	[237]
	[238]	[239]	[240]	[241]	[242]
	[243]	[244]	[245]	[246]	[247]
	[248]	[249]	[250]	[251]	[252]
	[253]	[254]			
2015	[255]	[256]	[257]	[258]	[259]
	[260]	[261]	[262]	[263]	[264]
	[265]	[266]	[267]	[268]	[269]
	[270]	[271]	[272]	[273]	[274]

	[275]	[276]	[277]	[278]	[279]
	[280]	[281]	[282]	[283]	[284]
	[285]	[286]	[287]	[288]	[289]
	[290]	[291]	[292]	[293]	
2016	[294]	[295]	[296]	[297]	[298]
	[299]	[300]	[301]	[302]	[303]
	[304]	[305]	[306]	[307]	[308]
	[309]	[310]	[311]	[312]	[313]
	[314]	[315]	[316]	[317]	[318]
	[319]	[320]	[321]	[322]	[323]
	[324]	[325]	[326]	[327]	[328]
	[329]	[330]	[331]	[332]	[333]
	[334]	[335]	[336]	[337]	[338]
	[339]	[340]	[341]	[342]	[343]
2017	[344]	[345]	[346]	[347]	[348]
	[349]	[350]	[351]	[352]	[353]
	[354]	[355]	[356]	[357]	[358]
	[359]	[360]	[361]	[362]	[363]
	[364]	[365]	[366]	[367]	

References

[1] C. Collberg, J. Nagra, Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection, Addison-Wesley Professional, 2009.
[2] E. Skoudis, L. Zeltser, Malware: Fighting Malicious Code, Prentice Hall Professional, Upper Saddle River, NJ 07458, 2004.
[3] B. Kitchenham, Procedures for performing systematic reviews, Technical Report TR/SE-0401, Department of Computer Science, Keele University, UK, 2004.
[4] R. Mallett, J. Hagen-Zanker, R. Slater, M. Duvendack, The benefits and challenges of using systematic reviews in international development research, J. Dev. Effectiveness 4 (3) (2012) 445–455.
[5] A. Balakrishnan, C. Schulze, Code obfuscation literature survey, Technical Report, University of Wisconsin, Madison, 2005.
[6] A. Majumdar, C. Thomborson, S. Drape, A survey of control-flow obfuscations, in: A. Bagchi, V. Atluri (Eds.), Information Systems Security, Lecture Notes in Computer Science, 4332 Springer Berlin Heidelberg, 2006, pp. 353–356.
[7] B. Kitchenham, P. Brereton, A systematic review of systematic review process research in software engineering, Inf. Softw. Technol. 55 (12) (2013) 2049–2075.
[8] IDA-PRO, (https://www.hex-rays.com/products/ida/). Accessed: 2018-07-01.
[9] J. Spradlin, Security through opcode randomization, 2012, US Patent App. 12/972,433.
[10] A. Main, M. Achim, S. Chow, H. Johnson, Y. Gu, Computer system protection by communication diversity, 2003, CA Patent App. CA 2,363,795.
[11] F.B. Cohen, Operating system protection through program evolution, Comput. Secur. 12 (6) (1993) 565–584.
[12] C. Pu, A. Black, C. Cowan, J. Walpole, C. Consel, A specialization toolkit to increase the diversity of operating systems, ICMAS Workshop Immunity-Based Systems, (1996).
[13] C. Collberg, C. Thomborson, D. Low, A taxonomy of obfuscating transformations, Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand, 1997.
[14] S. Forrest, A. Somayaji, D. Ackley, Building diverse computer systems, Operating Systems, The Sixth Workshop on Hot Topics, (1997), pp. 67–72.
[15] C. Collberg, C. Thomborson, D. Low, Breaking abstractions and unstructuring data structures, Computer Languages Proceedings. International Conference on, (1998), pp. 28–38.
[16] C. Collberg, C. Thomborson, D. Low, Manufacturing cheap, resilient, and stealthy opaque constructs, in: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, ACM, NY, USA, 1998, pp. 184–196.
[17] F. Hohl, Time limited blackbox security: protecting mobile agents from malicious hosts, in: G. Vigna (Ed.), Mobile Agents and Security, Lecture Notes in Computer Science, 1419 Springer Berlin Heidelberg, 1998, pp. 92–113.
[18] R.C. Linger, Systematic generation of stochastic diversity as an intrusion barrier in survivable systems software, Systems Sciences, HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on, IEEE, 1999, pp. 1–7.
[19] J. Hunt, Byte Code Protection, Java for Practitioners Practitioner Series, Springer London, 1999, pp. 427–429.
[20] V. Tam, R.K. Gupta, Using class decompilers to facilitate the security of Java applications!, Proceedings of the First International Conference on Web Information Systems Engineering. 1 (2000), pp. 153–158.
[21] C.C. Michael, A. Bartle, J. Viega, A. Hulot, N. Jarymowycz, J.R. Mills, B. Sohr, B. Arkin, Two systems for automatic software diversification, Proceedings of the DARPA Information Survivability Conference and Exposition, 2 IEEE Computer Society, 2000, p. 1220.
[22] C. Wang, J. Hill, J. Knight, J. Davidson, Software tamper resistance: obstructing static analysis of programs, Technical Report, University of Virginia, VA, USA, 2000.
[23] H. Goto, M. Mambo, K. Matsumura, H. Shizuya, Proceedings of the Information Security: Third International Workshop, ISW 2000 Wollongong, Australia, Proceedings, Springer Berlin Heidelberg, pp. 82–96.
[24] S. Chow, Y. Gu, H. Johnson, V. Zakharov, An approach to the obfuscation of control-flow of sequential computer programs, in: G. Davida, Y. Frankel (Eds.), Information Security, Lecture Notes in Computer Science, 2200 Springer Berlin Heidelberg, 2001, pp. 144–155.
[25] H. Goto, M. Mambo, H. Shizuya, Y. Watanabe, Evaluation of tamper-resistant software deviating from structured programming rules, in: V. Varadharajan, Y. Mu (Eds.), Information Security and Privacy, Lecture Notes in Computer Science, 2119 Springer Berlin Heidelberg, 2001, pp. 145–158.
[26] P. Shah, Code obfuscation for prevention of malicious reverse engineering attacks, 2002, A term paper for course ECE 578, Computer and Network Security, 12 pages.
[27] G. Wroblewski, General method of program code obfuscation, Ph.D. thesis, Wroclaw University, Poland, 2002.
[28] M. Chew, D. Song, Mitigating buffer overflows by operating system randomization, Technical Report CMU-CS-02-197, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2002.
[29] S. Drape, O. de Moor, G. Sittampalam, Transforming the .net intermediate language using path logic programming, in: Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '02, ACM, NY, USA, 2002, pp. 133–144.
[30] L. Badger, D. Kilpatrick, B. Matt, A. Reisse, T. Van Vleck, Self-protecting mobile agents obfuscation techniques evaluation report, Technical Report 01–036, NAI Labs, 2002.
[31] T. Ogiso, Y. Sakabe, M. Soshi, A. Miyaji, Software tamper resistance based on the difficulty of interprocedural analysis, Proceedings of the Interprocedural Analysis, 3rd International Workshop on Information Security Applications (WISA), (2002), pp. 437–452.
[32] C. Bain, D. Faatz, A. Fayad, D. Williams, Diversity as a defense strategy in information systems, in: M. Gertz, E. Guldentops, L. Strous (Eds.), Integrity, Internal Control and Security in Information Systems, IFIP – The International Federation for Information Processing, 83 Springer US, 2002, pp. 77–93.
[33] C.S. Collberg, C. Thomborson, Watermarking, tamper-proofing, and obfuscation - tools for software protection, IEEE Trans. Softw. Eng. 28 (8) (2002) 735–746.
[34] C. Linn, S. Debray, Obfuscation of executable code to improve resistance to static disassembly, Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS '03, NY, USA, 2003, pp. 290–299.
[35] D.C. DuVarney, V.N. Venkatakrishnan, S. Bhatkar, Self: a transparent security extension for elf binaries, Proceedings of the Workshop on New Security Paradigms, NSPW '03, ACM, USA, 2003, pp. 29–38.
[36] M. Sosonkin, G. Naumovich, N. Memon, Obfuscation of design intent in object-oriented applications, in: Proceedings of the 3rd ACM Workshop on Digital Rights Management, DRM '03, ACM, NY, USA, 2003, pp. 142–153.
[37] P. van Oorschot, Revisiting software protection, in: C. Boyd, W. Mao (Eds.), Information Security, Lecture Notes in Computer Science, 2851 Springer Berlin Heidelberg, 2003, pp. 1–13.
[38] Y. Kanzaki, A. Monden, M. Nakamura, K.-i. Matsumoto, Exploiting self-modification mechanism for program protection, Proceedings of the 27th Annual International Computer Software and Applications Conference, COMPSAC. Proceedings. (2003), pp. 170–179.
[39] E. Bhatkar, D.C. Duvarney, R. Sekar, Address obfuscation: an efficient approach to combat a broad range of memory error exploits, Proceedings of the 12th USENIX Security Symposium, (2003), pp. 105–120.
[40] L. D'Anna, B. Matt, A. Reisse, T.V. Vleck, S. Schwab, P. Leblanc, Self-protecting mobile agents obfuscation report, Technical Report 03–015, Network Associates

Laboratories, 2003.
[41] C. Collberg, G. Myles, A. Huntwork, Sandmark–a tool for software protection research, IEEE Secur. Priv. 1 (4) (2003) 40–49.
[42] G.S. Kc, A.D. Keromytis, V. Prevelakis, Countering code-injection attacks with instruction-set randomization, Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS '03, NY, USA, 2003, pp. 272–280.
[43] T. Ogiso, Y. Sakabe, M. Soshi, A. Miyaji, Software obfuscation on a theoretical basis and its implementation, IEICE Trans. Fundam. Electr., Commun. Comput. Sci. 86 (1) (2003) 176–186.
[44] Y. Sakabe, M. Soshi, A. Miyaji, Java obfuscation with a theoretical basis for building secure mobile agents, in: A. Lioy, D. Mazzocchi (Eds.), Communications and Multimedia Security. Advanced Techniques for Network and Data Protection, Lecture Notes in Computer Science, 2828 Springer Berlin Heidelberg, 2003, pp. 89–103.
[45] D. Rusu, Protection methods of Java bytecode, Proceedings of the Networking in Education and Research International Conference, (2003), pp. 214–220.
[46] J. Xu, Z. Kalbarczyk, R. Iyer, Transparent runtime randomization for security, Proceedings of the 22nd International Symposium on Reliable Distributed Systems. Proceedings. (2003), pp. 260–269.
[47] A. Monden, A. Monsifrot, C. Thomborson, Obfuscated instructions for software protection, Technical Report NAIST-IS-TR2003013, Graduate School of Information Science, Nara Institute of Science and Technology, Japan, 2003.
[48] C. Dahn, S. Mancoridis, Using program transformation to secure C programs against buffer overflows, Proceedings of the 10th Working Conference on Reverse Engineering, WCRE. IEEE, 2003, pp. 323–332.
[49] S. Ring, E. Cole, Taking a lesson from stealthy rootkits, IEEE Secur. Priv. 2 (4) (2004) 38–45.
[50] H. Saputra, G. Chen, R. Brooks, N. Vijaykrishnan, M. Kandemir, M.J. Irwin, Code protection for resource-constrained embedded devices, SIGPLAN Notices 39 (7) (2004) 240–248.
[51] J.T. Chan, W. Yang, Advanced obfuscation techniques for java bytecode, J. Syst. Softw. 71 (1–2) (2004) 1–10.
[52] L. Ertaul, S. Venkatesh, JHide–a tool kit for code obfuscation, Proceedings of the 8th IASTED International Conference on Software Engineering and Applications, (2004), pp. 133–138.
[53] W. Thompson, A. Yasinsac, J.T. McDonald, Semantic encryption transformation scheme, Proceedings of the International Workshop on Security in Parallel and Distributed Systems, PDCS, (2004), pp. 516–521.
[54] D.E. Bakken, R. Parameswaran, D.M. Blough, A.A. Franz, T.J. Palmer, Data obfuscation: anonymity and desensitization of usable data sets, IEEE Secur. Priv. 2 (6) (2004) 34–41.
[55] S. Boyd, A. Keromytis, Sqlrand: preventing Sql injection attacks, in: M. Jakobsson, M. Yung, J. Zhou (Eds.), Applied Cryptography and Network Security, Lecture Notes in Computer Science, 3089 Springer Berlin Heidelberg, 2004, pp. 292–302.
[56] S. Drape, Obfuscation of abstract data types, Ph.D. thesis, University of Oxford, UK, 2004.
[57] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, D. Boneh, On the effectiveness of address-space randomization, Proceedings of the 11th ACM Conference on Computer and Communications Security, CCS '04, NY, USA, 2004, pp. 298–307.
[58] K. Heffner, C. Collberg, The obfuscation executive, in: K. Zhang, Y. Zheng (Eds.), Information Security, Lecture Notes in Computer Science, 3225 Springer Berlin Heidelberg, 2004, pp. 428–440.
[59] D.E. Bakken, R. Parameswaran, D.M. Blough, A.A. Franz, T.J. Palmer, Data obfuscation: anonymity and desensitization of usable data sets, IEEE Secur. Priv. 2 (6) (2004) 34–41.
[60] A. Monden, A. Monsifrot, C. Thomborson, A framework for obfuscated interpretation, in: Proceedings of the 2nd Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation - Volume 32, ACSW Frontiers '04, Australian Computer Society, Inc., Darlinghurst, Australia, 2004, pp. 7–16.
[61] B. Anckaert, B. De Sutter, K. De Bosschere, Software piracy prevention through diversity, in: Proceedings of the 4th ACM Workshop on Digital Rights Management, DRM '04, ACM, NY, USA, 2004, pp. 63–71.
[62] J. Ge, S. Chaudhuri, A. Tyagi, Control flow based obfuscation, in: Proceedings of the 5th ACM Workshop on Digital Rights Management, DRM '05, ACM, NY, USA, 2005, pp. 83–92.
[63] L. Ertaul, S. Venkatesh, Novel obfuscation algorithms for software security, Proceedings of the International Conference on Software Engineering Research and Practice, SERP'05, (2005), pp. 209–215.
[64] C. Zhu, Z. Yin, A. Zhang, Mobile Code Security on Destination Platform, in: X. Lu, W. Zhao (Eds.), Networking and Mobile Computing, Lecture Notes in Computer Science, 3619 Springer Berlin Heidelberg, 2005, pp. 1263–1270.
[65] P.-Y. Chen, G. Kataria, R. Krishnan, Software diversity for information security. Proceedings of the Workshop on the Economics of Information Security (WEIS), Harvard University, Cambridge, MA, 2005, p. 20 pages.
[66] E.G. Barrantes, D.H. Ackley, S. Forrest, D. Stefanović, Randomized instruction set emulation, ACM Trans. Inf. Syst. Secur. 8 (1) (2005) 3–40.
[67] A. Majumdar, C. Thomborson, Securing mobile agents control flow using opaque predicates, in: R. Khosla, R. Howlett, L. Jain (Eds.), Knowledge-based intelligent information and engineering systems, Lecture Notes in Computer Science, 3683 Springer Berlin Heidelberg, 2005, pp. 1065–1071.
[68] J. Macbride, C. Mascioli, S. Marks, Y. Tang, L.M. Head, P. Ramach, A comparative study of Java obfuscators, Proceedings of the IASTED International Conference on Software Engineering and Applications (SEA), (2005), pp. 14–16.
[69] S. Bhatkar, R. Sekar, D.C. DuVarney, Efficient techniques for comprehensive protection from memory error exploits, Proceedings of the 14th Conference on USENIX Security Symposium - Vol. 14, SSYM'05, USENIX Association, Berkeley, CA, USA, 2005, p. 17.
[70] A.D. Keromytis, V. Prevelakis, A survey of randomization techniques against common mode attacks, Technical Report, Department of Computer Science, Drexel University, Philadelphia, Pennsylvania, USA, 2005.
[71] J.M. Memon, S. ul Arfeen, A. Mughal, F. Memon, Preventing reverse engineering threat in Java using byte code obfuscation techniques, Proceedings of the International Conference on Emerging Technologies. ICET '06. (2006), pp. 689–694.
[72] S. Drape, An obfuscation for binary trees, Proceedings of the TENCON. IEEE Region 10 Conference, IEEE, 2006, pp. 1–4.
[73] T. Hou, H. Chen, M. Tsai, Three control flow obfuscation methods for java software, IEEE Proc. Softw. 153 (6) (2006) 80–86.
[74] A. Majumdar, A. Monsifrot, C. Thomborson, On evaluating obfuscatory strength of alias-based transforms using static analysis, Proceedings of the International Conference on Advanced Computing and Communications, ADCOM. (2006), pp. 605–610.
[75] A. Majumdar, C. Thomborson, Manufacturing opaque predicates in distributed systems for code obfuscation, in: Proceedings of the 29th Australasian Computer Science Conference - Volume 48, ACSC '06, Australian Computer Society, Inc., Darlinghurst, Australia, 2006, pp. 187–196.
[76] Y. Kanzaki, A. Monden, M. Nakamura, K. Matsumoto, A software protection method based on instruction camouflage, Electr. Commun. Japan (Part III: Fundam. Electr. Sci.) 89 (1) (2006) 47–59.
[77] H. Yamauchi, Y. Kanzaki, A. Monden, M. Nakamura, K. Matsumoto, Software obfuscation from crackers' viewpoint. Proceedings of the ACST, (2006), pp. 286–291.
[78] M. Madou, B. Anckaert, B. De Bus, K. De Bosschere, J. Cappaert, B. Preneel, On the effectiveness of source code transformations for binary obfuscation, Proceedings of the International Conference on Software Engineering Research and Practice, SERP'06, CSREA Press, 2006, pp. 527–533.
[79] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, K. De Bosschere, Software protection through dynamic code mutation, in: J. Song, T. Kwon, M. Yung (Eds.), Information Security Applications, Lecture Notes in Computer Science, 3786 Springer Berlin Heidelberg, 2006, pp. 194–206.
[80] B. Anckaert, M. Jakubowski, R. Venkatesan, Proteus: Virtualization for diversified tamper-resistance, Proceedings of the ACM Workshop on Digital Rights Management, DRM '06, NY, USA, 2006, pp. 47–58.
[81] B. Cox, D. E., A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, J. Hiser, N-variant systems: a secretless framework for security through diversity. Usenix Security, 6 (2006), pp. 105–120.
[82] E. Totel, F. Majorczyk, L. Mé, Cots diversity based intrusion detection and application to web servers, in: A. Valdes, D. Zamboni (Eds.), Recent Advances in Intrusion Detection, Lecture Notes in Computer Science, 3858 Springer Berlin Heidelberg, 2006, pp. 43–62.
[83] R. Pucella, F.B. Schneider, Independence from obfuscation: a semantic framework for diversity, Proceedings of the 19th IEEE Computer Security Foundations Workshop, (2006), pp. 12 pages,–241.
[84] Y. Sakabe, M. Soshi, A. Miyaji, Java obfuscation approaches to construct tamper-resistant object-oriented programs, Inf. Media Technol. 1 (1) (2006) 134–146.
[85] W. Zhu, C. Thomborson, F.-Y. Wang, Obfuscate arrays by homomorphic functions, Proceedings of the IEEE International Conference on Granular Computing, (2006), pp. 770–773.
[86] K. Fukushima, S. Kiyomoto, T. Tanaka, An obfuscation scheme using affine transformation and its implementation, Inf. Media Technol. 1 (2) (2006) 1094–1108.
[87] M. Karnick, J. MacBride, S. McGinnis, Y. Tang, R. Ramachandran, A qualitative analysis of Java obfuscation, Proceedings of 10th IASTED International Conference on Software Engineering and Applications, TX, USA, 2006, pp. 166–171.
[88] C. Kil, J. Jim, C. Bookholt, J. Xu, P. Ning, Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software, Proceedings of the 22nd Annual Computer Security Applications Conference. ACSAC '06. (2006), pp. 339–348.
[89] J. Witkowska, Biometrics, Computer Security Systems and Artificial Intelligence Applications, Springer US, Boston, MA, pp. 175–182.
[90] M. Madou, Application security through program obfuscation, Ph.D. thesis, Ghent University, Belgium, 2006.
[91] B.D. Birrer, R.A. Raines, R.O. Baldwin, B.E. Mullins, R.W. Bennington, Program fragmentation as a metamorphic software protection, Proceedings of the Third International Symposium on Information Assurance and Security, IAS. (2007), pp. 369–374.
[92] S. Drape, A. Majumdar, C. Thomborson, Slicing aided design of obfuscating transforms, Proceedings of the 6th IEEE/ACIS International Conference on Computer and Information Science, ICIS. (2007), pp. 1019–1024.
[93] N. Naeem, M. Batchelder, L. Hendren, Metrics for measuring the effectiveness of decompilers and obfuscators, Proceedings of the 15th IEEE International Conference on Program Comprehension. ICPC '07. (2007), pp. 253–258.
[94] J. Wu, X. Guo, C. Tian, H. Yin, G. Chang, The study for protecting mobile agents based on time checking technology, Proceedings of the IEEE International Conference Robotics and Biomimetics. (2007), pp. 2013–2017.
[95] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, B. Preneel, Program obfuscation: a quantitative approach, Proceedings of the 2007 ACM Workshop on Quality of Protection, QoP '07, ACM, NY, USA, 2007, pp. 15–20.
[96] A. Majumdar, S.J. Drape, C.D. Thomborson, Slicing obfuscations: design,

correctness, and evaluation, Proceedings of ACM Workshop on Digital Rights Management, DRM '07, ACM, NY, USA, 2007, pp. 70–81.
[97] S. Drape, Generalising the array split obfuscation, Inf. Sci. (Ny) 177 (1) (2007) 202–219.
[98] Y. Kinoshita, K. Kashiwagi, Y. Higami, S.-Y. Kobayashi, Development of concealing the purpose of processing for programs in a distributed computing environment, in: J. Pejaś, K. Saeed (Eds.), Advances in Information Processing and Protection, Springer US, 2007, pp. 263–269.
[99] W.F. Zhu, Concepts and techniques in software watermarking and obfuscation, Ph.D. thesis, The University of Auckland, New Zealand, 2007.
[100] I.V. Popov, S.K. Debray, G.R. Andrews, Binary obfuscation using signals, Proceedings of the 16th USENIX Security Symposium on USENIX Security Symposium, SS'07, Berkeley, CA, USA, 2007, pp. 19:1–19:16.
[101] M. Batchelder, L. Hendren, Obfuscating Java: the most pain for the least gain, in: S. Krishnamurthi, M. Odersky (Eds.), Compiler Construction, Lecture Notes in Computer Science, 4420 Springer Berlin Heidelberg, 2007, pp. 96–110.
[102] S. Praveen, P.S. Lal, Array data transformation for source code obfuscation, Proceedings of the World Academy of Science, Engineering and Technology (PWASET) Volume, 21 (2007).
[103] B. Anckaert, M. Jakubowski, R. Venkatesan, K. De Bosschere, Run-time randomization to mitigate tampering, in: A. Miyaji, H. Kikuchi, K. Rannenberg (Eds.), Advances in Information and Computer Security, Lecture Notes in Computer Science, 4752 Springer Berlin Heidelberg, 2007, pp. 153–168.
[104] S. Drape, A. Majumdar, Design and evaluation of slicing obfuscation, Technical Report, Department of Computer Science, The University of Auckland, New Zealand, 2007.
[105] X. Jiang, H.J. Wang, D. Xu, Y. Wang, RandSys: thwarting code injection attacks with system service interface randomization, Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems, SRDS. (2007), pp. 209–218.
[106] D. Bruschi, L. Cavallaro, A. Lanzi, Diversified process replicæ for defeating memory error exploits, Proceedings of the IEEE International Performance, Computing, and Communications Conference. IPCCC 2007. (2007), pp. 434–441.
[107] N. Kisserli, J. Cappaert, B. Preneel, Software security through targeted diversification, Software Security Assessments–CoBaSSA, (2007). p. 10 pages.
[108] J.C. Knight, J.W. Davidson, D. Evans, A. Nguyen-Tuong, C. Wang, Genesis: a framework for achieving software component diversity, Technical Report, University of Virginia, Charlottesville VA, USA, 2007.
[109] X. Zhang, F. He, W. Zuo, An inter-classes obfuscation method for Java program, Proceedings of the International Conference on Information Security and Assurance, 2008. ISA 2008. IEEE, 2008, pp. 360–365.
[110] J. Qin, Z. Bai, Y. Bai, Polymorphic algorithm of JavaScript code protection, Proceedings of the International Symposium on Computer Science and Computational Technology, ISCSCT '08. 1 (2008), pp. 451–454.
[111] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella, Towards experimental evaluation of code obfuscation techniques, in: Proceedings of the 4th ACM Workshop on Quality of Protection, QoP '08, ACM, NY, USA, 2008, pp. 39–46.
[112] P. Sivadasan, P.S. Lal, Array based Java source code obfuscation using classes with restructured arrays, arXiv:0807.4309 (2008).
[113] S. Cho, H. Chang, Y. Cho, Implementation of an obfuscation tool for C/C++ source code protection on the XScale architecture, in: U. Brinkschulte, T. Givargis, S. Russo (Eds.), Software Technologies for Embedded and Ubiquitous Systems, Springer Berlin Heidelberg, 2008, pp. 406–416.
[114] M. Jacob, M. Jakubowski, P. Naldurg, C. Saw, R. Venkatesan, The Superdiversifier: peephole individualization for software protection, in: K. Matsuura, E. Fujisaki (Eds.), Advances in Information and Computer Security, Lecture Notes in Computer Science, 5312 Springer Berlin Heidelberg, 2008, pp. 100–120.
[115] D. Dolz, G. Parra, Using exception handling to build opaque predicates in intermediate code obfuscation techniques, J. Comput. Sci. Technol. 8 (2008).
[116] S. Bhatkar, R. Sekar, Data space randomization, in: D. Zamboni (Ed.), Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture Notes in Computer Science, 5137 Springer Berlin Heidelberg, 2008, pp. 1–22.
[117] H. Tamada, M. Nakamura, A. Monden, K. Matsumoto, Introducing dynamic name resolution mechanism for obfuscating system-defined names in programs, Proceedings of the International Conference on Software Engineering (IASTED SE), Innsbruck, Austria, 2008, pp. 125–130.
[118] A. Nguyen-Tuong, D. Evans, J. Knight, B. Cox, J. Davidson, Security through redundant data diversity, Proceedings of the IEEE International Conference Dependable Systems and Networks With FTCS and DCC. (2008), pp. 187–196.
[119] C. Cadar, P. Akritidis, M. Costa, J.-P. Martin, M. Castro, Data randomization, Technical Report, Microsoft Research, 2008.
[120] C. Liem, Y.X. Gu, H. Johnson, A compiler-based infrastructure for software-protection, Proceedings of the Third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS '08, ACM, NY, USA, 2008, pp. 33–44.
[121] H. Tamada, K. Fukuda, T. Yoshioka, Program incomprehensibility evaluation for obfuscation methods with queue-based mental simulation model, Proceedings of the 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel Distributed Computing (SNPD), (2012), pp. 498–503.
[122] K. Fukushima, S. Kiyomoto, T. Tanaka, Obfuscation mechanism in conjunction with tamper-proof module, Proceedings of the International Conference on Computational Science and Engineering. CSE '09. 2 (2009), pp. 665–670.
[123] S. Xiao, J. Park, Y. Ye, Tamper resistance for software defined radio software, Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference. COMPSAC '09. 1 (2009), pp. 383–391.
[124] H.-Y. Tsai, Y.-L. Huang, D. Wagner, A graph approach to quantitative analysis of control-flow obfuscating transformations, IEEE Trans. Inf. Forensics Secur. 4 (2) (2009) 257–267.
[125] Z. Tang, X. Chen, D. Fang, F. Chen, Research on Java software protection with the obfuscation in identifier renaming, Proceedings of the Fourth International Conference on Innovative Computing, Information and Control (ICICIC), (2009), pp. 1067–1071.
[126] D. Yi, A new obfuscation scheme in constructing fuzzy predicates, Proceedings of the WRI World Congress on Software Engineering. WCSE '09. 4 (2009), pp. 379–382.
[127] H. JingDe, S. Gang, Substitution encryption algorithm study for embedded mobile code protection, Proceedings of the International Conference on Communication Software and Networks, ICCSN '09. (2009), pp. 645–649.
[128] W. Jiehong, G. Fuxiang, Study of MA protection based extending inheritance hierarchy trees and time check, Proceedings of the 4th International Conference on Computer Science Education, ICCSE '09. (2009), pp. 380–384.
[129] O. Shevtsova, D. Buintsev, Methods and software for the program obfuscation, Proceedings of the International Siberian Conference on Control and Communications, SIBCON 2009. (2009), pp. 113–115.
[130] P. Sivadasan, P. SojanLal, N. Sivadasan, JDATATRANS for array obfuscation in Java source codes to defeat reverse engineering from decompiled codes, Proceedings of the 2Nd Bangalore Annual Compute Conference, COMPUTE '09, ACM, NY, USA, 2009, pp. 13:1–13:4.
[131] M. Ceccato, P. Tonella, M.D. Preda, A. Majumdar, Remote software protection by orthogonal client replacement, Proceedings of the ACM Symposium on Applied Computing, SAC '09, NY, USA, 2009, pp. 448–455.
[132] P. Wayner, Disappearing Cryptography: Information Hiding: Steganography & Watermarking, 3, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.
[133] Z. Lin, R. Riley, D. Xu, Polymorphing software by randomizing data structure layout, in: U. Flegel, D. Bruschi (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture Notes in Computer Science, 5587 Springer Berlin Heidelberg, 2009, pp. 107–126.
[134] B. De Sutter, B. Anckaert, J. Geiregat, D. Chanet, K. De Bosschere, Instruction set limitation in support of software diversity, in: P. Lee, J. Cheon (Eds.), Information Security and Cryptology ICISC 2008, Lecture Notes in Computer Science, 5461 Springer Berlin Heidelberg, 2009, pp. 152–165.
[135] M. Jakubowski, C. Saw, R. Venkatesan, Tamper-tolerant software: modeling and implementation, in: T. Takagi, M. Mambo (Eds.), Advances in Information and Computer Security, Lecture Notes in Computer Science, 5824 Springer Berlin Heidelberg, 2009, pp. 125–139.
[136] J. Han, D. Gao, R.H. Deng, On the effectiveness of software diversity: a systematic study on real-world vulnerabilities, in: U. Flegel, D. Bruschi (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Lecture Notes in Computer Science, 5587 Springer Berlin Heidelberg, 2009, pp. 127–146.
[137] S. Drape, I. Voiculescu, The use of matrices in obfuscation, Technical Report, Oxford University Computing Laboratory, Oxford, UK, 2009.
[138] P. Sivadasan, P.S. Lal, JConstHide: A Framework for Java Source Code Constant Hiding, CoRR (2009). arXiv:0904.3458
[139] B. Anckaert, M. Jakubowski, R. Venkatesan, C.W. Saw, Runtime protection via dataflow flattening, Proceedings of the 3rd International Conference on Emerging Security Information, Systems and Technologies. SECURWARE '09. (2009), pp. 242–248.
[140] D. Williams, W. Hu, J. Davidson, J. Hiser, J. Knight, A. Nguyen-Tuong, Security through diversity: leveraging virtual machine technology, IEEE Secur. Priv. 7 (1) (2009) 26–33.
[141] H. Xu, S.J. Chapin, Address-space layout randomization using code islands, J. Comput. Secur. 17 (3) (2009) 331–362.
[142] M.H. Jakubowski, C.W. Saw, R. Venkatesan, Iterated transformations and quantitative metrics for software protection, Proceedings of the International Conference on Security and Cryptography (SECRYPT), (2009), pp. 359–368.
[143] T. László, Á. Kiss, Obfuscating C++ programs via control flow flattening, Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica 30 (2009) 3–19.
[144] B. Coppens, I. Verbauwhede, K.D. Bosschere, B.D. Sutter, Practical mitigations for timing-based side-channel attacks on modern x86 processors, Proceedings of the 30th IEEE Symposium on Security and Privacy, (2009), pp. 45–60.
[145] S.M. Darwish, S.K. Guirguis, M.S. Zalat, Stealthy code obfuscation technique for software security, Proceedings of the International Conference on Computer Engineering and Systems (ICCES), IEEE, 2010, pp. 93–99.
[146] T. Long, L. Liu, Y. Yu, Z. Wan, Assure high quality code using refactoring and obfuscation techniques, Proceedings of the Fifth International Conference on Frontier of Computer Science and Technology (FCST), (2010), pp. 246–252.
[147] Z. Vrba, P. Halvorsen, C. Griwodz, Program obfuscation by strong cryptography, Proceedings of the International Conference on Availability, Reliability, and Security, ARES '10, (2010), pp. 242–247.
[148] X. Guangli, C. Zheng, The code obfuscation technology based on class combination, Proceedings of the Ninth International Symposium on Distributed Computing and Applications to Business Engineering and Science (DCABES), IEEE, 2010, pp. 479–483.
[149] Q. Gu, Efficient code diversification for network reprogramming in sensor networks, in: Proceedings of the Third ACM Conference on Wireless Network Security, WiSec '10, ACM, NY, USA, 2010, pp. 145–150.
[150] M. Franz, E Unibus Pluram: massive-scale software diversity as a defense mechanism, in: Proceedings of the Workshop on New Security Paradigms, NSPW '10, ACM, NY, USA, 2010, pp. 7–16.

[151] S.S. Yau, H.G. An, Protection of users' data confidentiality in cloud computing, in: Proceedings of the Second Asia-Pacific Symposium on Internetware, Internetware '10, ACM, NY, USA, 2010, pp. 11:1–11:6.
[152] B. Lee, Y. Kim, J. Kim, binob+: a framework for potent and stealthy binary obfuscation, in: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS '10, NY, USA, (2010), pp. 271–281.
[153] T. Roeder, F.B. Schneider, Proactive obfuscation, ACM Trans. Comput. Syst. 28 (2) (2010) 4:1–4:54.
[154] G. Portokalidis, A.D. Keromytis, Fast and practical instruction-set randomization for commodity systems, in: Proceedings of the 26th Annual Computer Security Applications Conference, ACSAC '10, ACM, NY, USA, 2010, pp. 41–48.
[155] X. Zhang, F. He, W. Zuo, Theory and practice of program obfuscation, in: M. Crisan (Ed.), Convergence and Hybrid Information Technologies, INTECH: Croatia, 2010, pp. 277–302.
[156] Y.Y. Wei, K. Ohzeki, Obfuscation methods with controlled calculation amounts and table function, Proceedings of the International Multiconference on Computer Science and Information Technology (IMCSIT), (2010), pp. 775–780.
[157] J. Cappaert, B. Preneel, A general model for hiding control flow, in: Proceedings of the Tenth Annual ACM Workshop on Digital Rights Management, DRM '10, ACM, NY, USA, 2010, pp. 35–42.
[158] S.W. Boyd, G.S. Kc, M.E. Locasto, A.D. Keromytis, V. Prevelakis, On the general applicability of instruction-set randomization, IEEE Trans. Dependable Secure Comput. 7 (3) (2010) 255–270.
[159] F. Baiardi, D. Sgandurra, An obfuscation-based approach against injection attacks, Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES), (2011), pp. 51–58.
[160] P. Falcarin, S. Carlo, A. Cabutto, N. Garazzino, D. Barberis, Exploiting code mobility for dynamic binary obfuscation, Proceedings of the World Congress on Internet Security (WorldCIS), (2011), pp. 114–120.
[161] N. Mavrogiannopoulos, N. Kisserli, B. Preneel, A taxonomy of self-modifying code for obfuscation, Comput. Secur. 30 (8) (2011) 679–691.
[162] M. Heiderich, N. Eduardo Alberto Vela, G. Heyes, D. Lindsay, Web Application Obfuscation, Elsevier, Boston, USA, 2011.
[163] M. Christodorescu, M. Fredrikson, S. Jha, J. Giffin, End-to-end software diversification of internet services, in: S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, X.S. Wang (Eds.), Moving Target Defense, Advances in Information Security, 54 Springer NY, 2011, pp. 117–130.
[164] R. Chakraborty, S. Narasimhan, S. Bhunia, Embedded software security through key-based control flow obfuscation, in: M. Joye, D. Mukhopadhyay, M. Tunstall (Eds.), Security Aspects in Information Technology, Lecture Notes in Computer Science, 7011 Springer Berlin Heidelberg, 2011, pp. 30–44.
[165] L. Shan, S. Emmanuel, Mobile agent protection with self-modifying code, J. Signal Process. Syst. 65 (1) (2011) 105–116.
[166] A. Amarilli, S. Müller, D. Naccache, D. Page, P. Rauzy, M. Tunstall, Can code polymorphism limit information leakage? in: C. Ardagna, J. Zhou (Eds.), Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication, Lecture Notes in Computer Science, 6633 Springer Berlin Heidelberg, 2011, pp. 1–21.
[167] G. Portokalidis, A. Keromytis, Global ISR: toward a comprehensive defense against unauthorized code execution, in: S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, X.S. Wang (Eds.), Moving Target Defense, Advances in Information Security, 54 Springer NY, 2011, pp. 49–76.
[168] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, M. Franz, Compiler-generated software diversity, in: S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, X.S. Wang (Eds.), Moving Target Defense, Advances in Information Security, 54 Springer NY, 2011, pp. 77–98.
[169] M. Ceccato, P. Tonella, Codebender: remote software protection using orthogonal replacement, IEEE Softw. 28 (2) (2011) 28–34.
[170] S. Armoogum, A. Caully, Obfuscation techniques for mobile agent code confidentiality, J. Inf. Syst. Manag. 1 (1) (2011) 83–94.
[171] P. Sivadasan, P.S. Lal, Suggesting potency measures for obfuscated arrays and usage of source code obfuscators for intellectual property protection of Java products, Proceedings of the International Conference on Information and Network Technology (ICINT), (2011).
[172] Y. Huang, A.K. Ghosh, Introducing diversity and uncertainty to create moving attack surfaces for web services, in: S. Jajodia, K.A. Ghosh, V. Swarup, C. Wang, S.X. Wang (Eds.), Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats, Springer NY, NY, 2011, pp. 131–151.
[173] D. Evans, A. Nguyen-Tuong, J. Knight, Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats, Springer NY, NY, pp. 29–48.
[174] J.C. Knight, Dependable and Historic Computing: Essays dedicated To Brian Randell on the Occasion of his 75th Birthday, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 298–312.
[175] X. Yao, J. Pang, Y. Zhang, Y. Yu, J. Lu, A method and implementation of control flow obfuscation using SEH, Proceedings of the 4th International Conference on Multimedia Information Networking and Security (MINES), (2012), pp. 336–339.
[176] A. Capiluppi, P. Falcarin, C. Boldyreff, Code defactoring: evaluating the effectiveness of Java obfuscations, Proceedings of the 19th Working Conference on Reverse Engineering (WCRE), (2012), pp. 71–80.
[177] Y. Le, H. Huo-Jiao, Research on Java bytecode parse and obfuscate tool, Proceedings of the International Conference on Computer Science Service System (CSSS), (2012), pp. 50–53.
[178] B. Rodes, Stack layout transformation: towards diversity for securing binary programs, Proceedings of the 34th International Conference on Software Engineering (ICSE), (2012), pp. 1543–1546.
[179] Z. Wang, C. Jia, M. Liu, X. Yu, Branch obfuscation using code mobility and signal, Proceedings of the IEEE 36th Annual Computer Software and Applications Conference Workshops (COMPSACW), (2012), pp. 553–558.
[180] M. Hataba, A. El-Mahdy, Cloud protection by obfuscation: techniques and metrics, Proceedings of the 2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), (2012), pp. 369–372.
[181] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella, The effectiveness of source code obfuscation: an experimental assessment, Proceedings of the IEEE 17th International Conference on Program Comprehension, ICPC '09. (2009), pp. 178–187.
[182] A. Zambon, Aucsmith-like obfuscation of Java bytecode, Proceedings of the IEEE 12th International Working Conference on Source Code Analysis and Manipulation (SCAM), (2012), pp. 114–119.
[183] S. Qing, W. Zhi-yue, W. Wei-min, L. Jing-liang, H. Zhi-wei, Technique of source code obfuscation based on data flow and control flow transformations, Proceedings of the 7th International Conference on Computer Science Education (ICCSE), (2012), pp. 1093–1097.
[184] D. Clarke, P. Ezhilchelvan, Fortress: adding intrusion-resilience to primary-backup server systems, Proceedings of the IEEE 31st Symposium on Reliable Distributed Systems (SRDS), (2012), pp. 121–130.
[185] M. Palanques, R. Dipietro, C.d. Ojo, M. Malet, M. Marino, T. Felguera, Secure cloud browser: model and architecture to support secure web navigation, in: Proceedings of the IEEE 31st Symposium on Reliable Distributed Systems, SRDS '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 402–403.
[186] R. Wu, P. Chen, B. Mao, L. Xie, RIM: a method to defend from JIT spraying attack, Proceedings of the Seventh International Conference on Availability, Reliability and Security (ARES), (2012), pp. 143–148.
[187] Y. Park, S.J. Stolfo, Software decoys for insider threat, in: Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security, ASIACCS '12, ACM, NY, USA, 2012, pp. 93–94.
[188] R. Giacobazzi, N.D. Jones, I. Mastroeni, Obfuscation by partial evaluation of distorted interpreters, Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '12, NY, USA, 2012, pp. 63–72.
[189] C. LeDoux, M. Sharkey, B. Primeaux, C. Miles, Instruction embedding for improved obfuscation, Proceedings of the 50th Annual Southeast Regional Conference, ACM-SE '12, ACM, NY, USA, 2012, pp. 130–135.
[190] Y.-L. Huang, H.-Y. Tsai, A framework for quantitative evaluation of parallel control-flow obfuscation, Comput. Secur. 31 (8) (2012) 886–896.
[191] P. Sivadasan, P.S. Lal, Securing SQLJ source codes from business logic disclosure by data hiding obfuscation, CoRR (2012). arXiv:1205.4813
[192] R. Wartell, V. Mohan, K.W. Hamlen, Z. Lin, Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code, Proceedings of the ACM Conference on Computer and Communications Security, CCS '12, NY, USA, 2012, pp. 157–168.
[193] M. Abadi, G.D. Plotkin, On protection by layout randomization, ACM Trans. Inf. Syst. Secur. 15 (2) (2012).
[194] C. Giuffrida, A. Kuijsten, A.S. Tanenbaum, Enhanced operating system security through efficient and fine-grained address space randomization, Presented as part of the 21st USENIX Security Symposium (USENIX Security 12), USENIX, Bellevue, WA, 2012, pp. 475–490.
[195] V. Pappas, M. Polychronakis, A. Keromytis, Smashing the gadgets: Hindering return-oriented programming using in-place code randomization, Proceedings of the IEEE Symposium on Security and Privacy (SP), (2012), pp. 601–615.
[196] L. Ďurfina, D. Kolář, C source code obfuscator, Kybernetika 48 (3) (2012) 494–501.
[197] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, J.W. Davidson, Ilr: where'd my gadgets go?, Proceedings of the IEEE Symposium on Security and Privacy (SP), (2012), pp. 571–585.
[198] R. Costa, L. Pirmez, D. Boccardo, L.F. Rust, R. Machado, TinyObf: code obfuscation framework for wireless sensor networks, Proceedings of the International Conference on Wireless Networks (ICWN), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012, pp. 68–74.
[199] S. Rauti, V. Leppänen, Browser extension-based man-in-the-browser attacks against Ajax applications with countermeasures, Proceedings of the 13th International Conference on Computer Systems and Technologies, CompSysTech '12, ACM, NY, USA, 2012, pp. 251–258.
[200] C. Collberg, S. Martin, J. Myers, J. Nagra, Distributed application tamper detection via continuous software updates, Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC '12, ACM, NY, USA, 2012, pp. 319–328.
[201] B. Bertholon, S. Varrette, S. Martinez, ShadObf: a C-source obfuscator based on multi-objective optimisation algorithms, Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), (2013), pp. 435–444.
[202] V. Balachandran, S. Emmanuel, Potent and stealthy control flow obfuscation by stack based self-modifying code, IEEE Trans. Inf. Forensics Secur. 8 (4) (2013) 669–681.
[203] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, M. Franz, Profile-guided automated software diversity, Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO), (2013), pp. 1–11.
[204] B. Coppens, B. De Sutter, K. De Bosschere, Protecting your software updates, IEEE Secur. Priv. 11 (2) (2013) 47–54.
[205] D. Dunaev, L. Lengyel, Aspects of intermediate level obfuscation, Proceedings of the 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC), (2013), pp. 138–142.
[206] A. Gupta, M. Kirkpatrick, E. Bertino, A secure architecture design based on application isolation, code minimization and randomization, Proceedings of the IEEE

Conference on Communications and Network Security (CNS), (2013), pp. 423–429.
[207] D. Stanley, D. Xu, E. Spafford, Improved kernel security through memory layout randomization, Proceedings of the IEEE 32nd International Performance Computing and Communications Conference (IPCCC), (2013), pp. 1–10.
[208] P. Chen, R. Wu, B. Mao, JITSafe: a framework against just-in-time spraying attacks, IET Inf. Secur. 7 (9) (2013) 283–292.
[209] B. Coppens, B. De Sutter, J. Maebe, Feedback-driven binary code diversification, ACM Trans. Archit. Code Optim. 9 (4) (2013) 24:1–24:26.
[210] D.d.A.H. Marco, I. Ripoll, J.C. Ruiz, Security through emulation-based processor diversification, in: B. Akhgar, H.R. Arabnia (Eds.), Proceedings of the Emerging Trends in ICT Security, 1st edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013.
[211] V. Balachandran, S. Emmanuel, Software protection with obfuscation and encryption, in: R. Deng, T. Feng (Eds.), Proceedings of the Information Security Practice and Experience, Lecture Notes in Computer Science, 7863 Springer Berlin Heidelberg, 2013, pp. 309–320.
[212] V. Samawi, A. Sulaiman, Software protection via hiding function using software obfuscation. Int. Arab J. Inf. Technol. 10 (6) (2013) 587–594.
[213] A. Kovacheva, Efficient code obfuscation for android, in: B. Papasratorn, N. Charoenkitkarn, V. Vanijja, V. Chongsuphajaisiddhi (Eds.), Advances in Information Technology, Communications in Computer and Information Science, 409 Springer International Publishing, 2013, pp. 104–119.
[214] V. Pappas, M. Polychronakis, A. Keromytis, Practical software diversification using in-place code randomization, in: S. Jajodia, A.K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, X.S. Wang (Eds.), Moving Target Defense II, Advances in Information Security, 100 Springer NY, 2013, pp. 175–202.
[215] T. Jackson, A. Homescu, S. Crane, P. Larsen, S. Brunthaler, M. Franz, Diversifying the software stack using randomized NOP insertion, in: S. Jajodia, A.K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, X.S. Wang (Eds.), Moving Target Defense II, Advances in Information Security, 100 Springer NY, 2013, pp. 151–173.
[216] A. Gupta, S. Kerr, M. Kirkpatrick, E. Bertino, Marlin: a fine grained randomization approach to defend against rop attacks, in: J. Lopez, X. Huang, R. Sandhu (Eds.), Network and System Security, Lecture Notes in Computer Science, 7873 Springer Berlin Heidelberg, 2013, pp. 293–306.
[217] M. Stewart, Algorithmic diversity for software security, CoRR (2013). arXiv:1312.3891
[218] S. Crane, P. Larsen, S. Brunthaler, M. Franz, Booby trapping software, in: Proceedings of the 2013 Workshop on New Security Paradigms Workshop, NSPW '13, ACM, NY, USA, 2013, pp. 95–106.
[219] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, W. Zou, Practical control flow integrity and randomization for binary executables, Proceedings of the IEEE Symposium on Security and Privacy (SP), (2013), pp. 559–573.
[220] M. Kanter, S. Taylor, Diversity in cloud systems through runtime and compile-time relocation, Proceedings of the IEEE International Conference on Technologies for Homeland Security (HST), (2013), pp. 396–402.
[221] L.V. Davi, A. Dmitrienko, S. Nürnberger, A.-R. Sadeghi, Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm, Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS '13, ACM, NY, USA, 2013, pp. 299–310.
[222] A. Homescu, S. Brunthaler, P. Larsen, M. Franz, Librando: transparent code randomization for just-in-time compilers, Proceedings of the ACM SIGSAC Conference on Computer & Communications Security, CCS '13, ACM, NY, USA, 2013, pp. 993–1004.
[223] C. Foket, B.D. Sutter, K.D. Bosschere, Pushing java type obfuscation to the limit, IEEE Trans. Dependable Secure Comput. 11 (6) (2014) 553–567.
[224] P. Larsen, S. Brunthaler, M. Franz, Security through diversity: are we there yet?, IEEE Secur. Priv. 12 (2) (2014) 28–35.
[225] S. Blazy, S. Riaud, Measuring the robustness of source program obfuscation: Studying the impact of compiler optimizations on the obfuscation of C programs, Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, CODASPY '14, ACM, NY, USA, 2014, pp. 123–126.
[226] K. Lu, S. Xiong, D. Gao, Ropsteg: program steganography with return oriented programming, Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, CODASPY '14, ACM, NY, USA, 2014, pp. 265–272.
[227] H.-T. Liaw, S.-C. Wei, Obfuscation for object-oriented programs: dismantling instance methods, Softw.: Pract. Exp. 44 (9) (2014) 1077–1104.
[228] M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, P. Tonella, A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques, Empir. Softw. Eng. 19 (4) (2014) 1040–1074.
[229] A. Kulkarni, R. Metta, A code obfuscation framework using code clones, in: Proceedings of the 22nd International Conference on Program Comprehension, ICPC, ACM, NY, USA, 2014, pp. 295–299.
[230] T. Tamboli, T.H. Austin, M. Stamp, Metamorphic code generation from llvm bytecode, J. Comput. Virology Hacking Tech. 10 (3) (2014) 177–187.
[231] A. Kulkarni, R. Metta, A new code obfuscation scheme for software protection, Proceedings of the IEEE 8th International Symposium on Service Oriented System Engineering (SOSE), (2014), pp. 409–414.
[232] S. Han, M. Ryu, J. Cha, B.U. Choi, Hotdol: Html obfuscation with text distribution to overlapping layers, Proceedings of the IEEE International Conference on Computer and Information Technology (CIT), (2014), pp. 399–404.
[233] S. Laurén, P. Mäki, S. Rauti, S. Hosseinzadeh, S. Hyrynsalmi, V. Leppänen, Symbol diversification of linux binaries, Proceedings of the World Congress on Internet Security (WorldCIS), IEEE, 2014, pp. 74–79.
[234] Y. Zhuang, M. Protsenko, T. Muller, F. Freiling, An(other) exercise in measuring the strength of source code obfuscation, Proceedings of the 25th International Workshop on Database and Expert Systems Applications (DEXA), (2014), pp. 313–317.
[235] L. Arockiam, S. Monikandan, Efficient cloud storage confidentiality to ensure data security, Proceedings of the International Conference on Computer Communication and Informatics (ICCCI), (2014), pp. 1–5.
[236] C. Tunc, F. Fargo, Y. Al-Nashif, S. Hariri, J. Hughes, Autonomic resilient cloud management (ARCM) design and evaluation, Proceedings of the International Conference Cloud and Autonomic Computing (ICCAC), (2014), pp. 44–49.
[237] M. Murphy, P. Larsen, S. Brunthaler, M. Franz, Software profiling options and their effects on security based diversification, in: Proceedings of the First ACM Workshop on Moving Target Defense, MTD '14, ACM, NY, USA, 2014, pp. 87–96.
[238] J. Seibert, H. Okhravi, E. Söderström, Information leaks without memory disclosures: Remote side channel attacks on diversified code, Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS '14, ACM, NY, USA, 2014, pp. 54–65.
[239] R. Omar, A. El-Mahdy, E. Rohou, Arbitrary control-flow embedding into multiple threads for obfuscation: a preliminary complexity and performance analysis, in: Proceedings of the 2Nd International Workshop on Security in Cloud Computing, SCC '14, ACM, NY, USA, 2014, pp. 51–58.
[240] S. Schrittwieser, S. Katzenbeisser, Code Obfuscation Against Static and Dynamic Reverse Engineering, Springer, Berlin, Heidelberg, pp. 270–284.
[241] X. Xie, F. Liu, B. Lu, A data obfuscation based on state transition graph of mealy automata, in: D.-S. Huang, V. Bevilacqua, P. Premaratne (Eds.), Intelligent Computing Theory, Lecture Notes in Computer Science, 8588 Springer International Publishing, 2014, pp. 520–531.
[242] H. Fang, Y. Wu, S. Wang, Y. Huang, Multi-stage binary code obfuscation using improved virtual machine, in: X. Lai, J. Zhou, H. Li (Eds.), Information Security, Lecture Notes in Computer Science, 7001 Springer Berlin Heidelberg, 2011, pp. 168–181.
[243] H. Okhravi, J. Riordan, K. Carter, Quantitative evaluation of dynamic platform techniques as a defensive mechanism, in: A. Stavrou, H. Bos, G. Portokalidis (Eds.), Research in Attacks, Intrusions and Defenses, Lecture Notes in Computer Science, 8688 Springer International Publishing, 2014, pp. 405–425.
[244] C. Huang, S. Zhu, R. Erbacher, Toward software diversity in heterogeneous networked systems, in: V. Atluri, G. Pernul (Eds.), Data and Applications Security and Privacy XXVIII, Lecture Notes in Computer Science, 8566 Springer Berlin Heidelberg, 2014, pp. 114–129.
[245] V. Balachandran, N.W. Keong, S. Emmanuel, Function level control flow obfuscation for software security, Proceedings of the 8th International Conference on Complex, Intelligent and Software Intensive Systems (CISIS), (2014), pp. 133–140.
[246] C. Sahin, P. Tornquist, R. Mckenna, Z. Pearson, J. Clause, How does code obfuscation impact energy usage? Proceedings of the IEEE International Conference Software Maintenance and Evolution (ICSME), (2014), pp. 131–140.
[247] V. Balachandran, S. Emmanuel, N.W. Keong, Obfuscation by code fragmentation to evade reverse engineering, Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC), (2014), pp. 463–469.
[248] P. Larsen, A. Homescu, S. Brunthaler, M. Franz, SoK: automated software diversity, Proceedings of the IEEE Symposium on Security and Privacy (SP), (2014), pp. 276–291.
[249] M. Backes, S. Nürnberger, Oxymoron: making fine-grained memory randomization practical by allowing code sharing, Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, 2014, pp. 433–447.
[250] B. Baudry, S. Allier, M. Monperrus, Tailored source code transformations to synthesize computationally diverse program variants, in: Proceedings of the International Symposium on Software Testing and Analysis, (ISSTA), ACM, NY, USA, 2014, pp. 149–159.
[251] H. Okhravi, T. Hobson, D. Bigelow, W. Streilein, Finding focus in the blur of moving-target techniques, IEEE Secur. Priv. 12 (2) (2014) 16–26.
[252] S. Rauti, J. Holvitie, V. Leppänen, Towards a diversification framework for operating system protection, in: Proceedings of the 15th International Conference on Computer Systems and Technologies, CompSysTech '14, ACM, NY, USA, 2014, pp. 286–293.
[253] S. Rauti, V. Leppänen, A proxy-like obfuscator for web application protection, Int. J. Inf. Technol. Secur. 6 (1) (2014) 39–52.
[254] S. Rauti, S. Laurén, S. Hosseinzadeh, J.-M. Mäkelä, S. Hyrynsalmi, V. Leppänen, Proceedings of the Trusted Systems: 6th International Conference, INTRUST, Beijing, China, Springer International Publishing, Cham, pp. 15–35.
[255] M. Ceccato, A. Capiluppi, P. Falcarin, C. Boldyreff, A large study on the effect of code obfuscation on the quality of java code, Empir. Softw. Eng. 20 (6) (2015) 1486–1524.
[256] A.R. Nurmukhametov, S.F. Kurmangaleev, V.V. Kaushan, S.S. Gaissaryan, Application of compiler transformations against software vulnerabilities exploitation, Progr. Comput. Softw. 41 (4) (2015) 231–236.
[257] P. Laperdrix, W. Rudametkin, B. Baudry, Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification, Proceedings of the IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), (2015), pp. 98–108.
[258] K.J. Hole, Toward anti-fragility: a malware-halting technique, IEEE Secur. Priv. 13 (4) (2015) 40–46.
[259] S. Ghosh, J. Hiser, J. Davidson, Matryoshka: Strengthening software protection via nested virtual machines, Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO), (2015), pp. 10–16.
[260] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, M. Franz, Readactor: practical code randomization resilient to memory disclosure, Proceedings of the IEEE Symposium on Security and Privacy (SP), (2015), pp.

763–780.
[261] Y. Wang, J. Wei, Toward protecting control flow confidentiality in cloud-based computation, Comput. Secur. 52 (2015) 106–127.
[262] K. Hole, Diversity reduces the impact of malware, IEEE Secur. Priv. 13 (3) (2015) 48–54.
[263] L. Davi, C. Liebchen, A.-R. Sadeghi, K.Z. Snow, F. Monrose, Isomeron: code randomization resilient to (just-in-time) return-oriented programming., 2015.
[264] V. Mohan, P. Larsen, S. Brunthaler, K.W. Hamlen, M. Franz, Opaque control-flow integrity, Proceedings of the NDSS, 26 (2015), pp. 27–30.
[265] A. Höller, T. Rauter, J. Iber, C. Kreiner, Proceedings of the 7th International Workshop Software Engineering for Resilient Systems: SERENE, Proceedings, Paris, FranceSpringer International Publishing, Cham, pp. 16–30.
[266] M. Protsenko, T. Müller, Trust, Privacy and Security in Digital Business: Trustbus, Valencia, Spain, Springer International Publishing, Cham, pp. 99–110.
[267] F. Nasim, B. Aslam, W. Ahmed, T. Naeem, Proceedings of the First International Conference Codes, Cryptology, and Information Security: C2SI Morocco, Proceedings - In honor of T. Berger, Springer International Publishing, Cham, pp. 297–313.
[268] R. Fedler, S. Banescu, A. Pretschner, Proceedings of the 34th International Conference Computer Safety, Reliability, and Security: SAFECOMP, delft, The Netherlands, Proceedings, Springer International Publishing, Cham, pp. 362–371.
[269] H.P. Joshi, A. Dhanasekaran, R. Dutta, Impact of software obfuscation on susceptibility to return-oriented programming attacks, Proceedings of the 36th IEEE Sarnoff Symposium, (2015), pp. 161–166.
[270] S. Allier, O. Barais, B. Baudry, J. Bourcier, E. Daubert, F. Fleurey, M. Monperrus, H. Song, M. Tricoire, Multitier diversification in web-based software applications, IEEE Softw. 32 (1) (2015) 83–90.
[271] P. Junod, J. Rinaldini, J. Wehrli, J. Michielin, Obfuscator-LLVM: software protection for the masses, Proceedings of the 1st International Workshop on Software Protection, SPRO '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 3–9.
[272] M. Hataba, R. Elkhouly, A. El-Mahdy, Diversified remote code execution using dynamic obfuscation of conditional branches, Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW), (2015), pp. 120–127.
[273] B.F. Demissie, M. Ceccato, R. Tiella, Assessment of data obfuscation with residue number coding, in: Proceedings of the 1st International Workshop on Software Protection, SPRO '15, IEEE, NJ, USA, 2015, pp. 38–44.
[274] P. Larsen, S. Brunthaler, M. Franz, Automatic software diversity, IEEE Secur. Priv. 13 (2) (2015) 30–37.
[275] S. Hosseinzadeh, S. Rauti, S. Hyrynsalmi, V. Leppänen, Security in the Internet of Things through obfuscation and diversification, Proceedings of the International Conference on Computing, Communication and Security (ICCCS), (2015), pp. 1–5.
[276] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, M. Franz, Large-scale automated software diversity–program evolution redux, IEEE Trans. Dependable Secure Comput. 14 (2) (2015) 158–171.
[277] S. Blazy, S. Riaud, T. Sirvent, Data tainting and obfuscation: improving plausibility of incorrect taint, Proceedings of the IEEE 15th International Working Conference Source Code Analysis and Manipulation (SCAM), (2015), pp. 111–120.
[278] E. Avidan, D.G. Feitelson, From obfuscation to comprehension, in: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 178–181.
[279] B. Abrath, B. Coppens, S. Volckaert, B.D. Sutter, Obfuscating windows DLLs, Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO), (2015), pp. 24–30.
[280] M. Protsenko, S. Kreuter, T. Müller, Dynamic self-protection and tamperproofing for android apps using native code, Proceedings of the 10th International Conference Availability, Reliability and Security (ARES), (2015), pp. 129–138.
[281] Y. Kanzaki, C. Thomborson, A. Monden, C. Collberg, Pinpointing and hiding surprising fragments in an obfuscated program, in: Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW-5, ACM, NY, USA, 2015, pp. 8:1–8:9.
[282] S. Laurén, S. Rauti, V. Leppänen, Diversification of system calls in linux kernel, in: Proceedings of the 16th International Conference on Computer Systems and Technologies, CompSysTech '15, ACM, NY, USA, 2015, pp. 284–291.
[283] A. Jangda, M. Mishra, B. De Sutter, Adaptive just-in-time code diversification, in: Proceedings of the Second ACM Workshop on Moving Target Defense, MTD '15, ACM, NY, USA, 2015, pp. 49–53.
[284] B. Baudry, M. Monperrus, The multiple facets of software diversity: recent developments in year 2000 and beyond, ACM Comput. Surv. 48 (1) (2015) 16:1–16:26.
[285] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, M. Franz, Thwarting cache side-channel attacks through dynamic software diversity, Proceedings of the NDSS, (2015), pp. 8–11.
[286] B. Baudry, S. Allier, M. Rodriguez-Cancio, M. Monperrus, Automatic software diversity in the light of test suites, CoRR (2015). arXiv:1509.00144
[287] B. Tello, M. Winterrose, G. Baah, M. Zhivich, Simulation based evaluation of a code diversification strategy, Proceedings of the 5th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH), (2015), pp. 36–43.
[288] C.K. Behera, D.L. Bhaskari, Different obfuscation techniques for code protection, Procedia Comput. Sci. 70 (2015) 757–763.
[289] S. Banescu, M. Ochoa, A. Pretschner, A framework for measuring software obfuscation resilience against automated attacks, Proceedings of the IEEE/ACM 1st International Workshop on Software Protection (SPRO), (2015), pp. 45–51.
[290] M. Hataba, A. El-Mahdy, E. Rohou, OJIT: a novel obfuscation approach using standard just-in-time compiler transformations, Proceedings of the International Workshop on Dynamic Compilation Everywhere, Netherlands, 2015.
[291] S. Banescu, A. Pretschner, D. Battré, S. Cazzulani, R. Shield, G. Thompson, Software-based protection against changeware, in: Proceedings of the 5th ACM Conference on Data and Application Security and Privacy, CODASPY '15, ACM, NY, USA, 2015, pp. 231–242.
[292] B. Baudry, S. Allier, M. Rodriguez-Cancio, M. Monperrus, DSpot: test amplification for automatic assessment of computational diversity, CoRR (2015). arXiv:1503.05807
[293] S. Hosseinzadeh, S. Hyrynsalmi, M. Conti, V. Leppänen, Security and privacy in cloud computing via obfuscation and diversification: a survey, Proceedings of the IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2015, pp. 529–535.
[294] Y. Wu, V. Suhendra, H. Saputra, Z. Zhao, Obfuscating software puzzle for denial-of-service attack mitigation, Proceedings of the IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), IEEE, 2016, pp. 115–122.
[295] A. Viticchié, L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella, R. Tiella, Assessment of source code obfuscation techniques, Proceedings of the IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2016, pp. 11–20.
[296] Z. Yujia, P. Jianmin, A new compile-time obfuscation scheme for software protection, Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), IEEE, 2016, pp. 1–5.
[297] K. Mahmood, D.M. Shila, Moving target defense for internet of things using context aware code partitioning and code diversification, Proceedings of the IEEE 3rd World Forum on Internet of Things (WF-IoT), IEEE, 2016, pp. 329–330.
[298] M. Styugin, V. Zolotarev, A. Prokhorov, R. Gorbil, New approach to software code diversification in interpreted languages based on the moving target technology, Proceedings of the IEEE 10th International Conference on Application of Information and Communication Technologies (AICT), IEEE, 2016, pp. 1–5.
[299] J. Gionta, W. Enck, P. Larsen, Preventing kernel code-reuse attacks through disclosure resistant code diversification, Proceedings of the IEEE Conference on Communications and Network Security (CNS), IEEE, 2016, pp. 189–197.
[300] H. Liu, Towards better program obfuscation: optimization via language models, Proceedings of the 38th International Conference on Software Engineering Companion, ICSE '16, ACM, New York, NY, USA, 2016, pp. 680–682.
[301] X. Xie, B. Lu, D. Gong, X. Luo, F. Liu, Random table and hash coding-based binary code obfuscation against stack trace analysis, IET Inf. Secur. 10 (1) (2016) 18–27.
[302] S.A. Sebastian, S. Malgaonkar, P. Shah, M. Kapoor, T. Parekhji, A study review on code obfuscation, Proceedings of the World Conference on Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave), IEEE, 2016, pp. 1–6.
[303] K. Kuang, Z. Tang, X. Gong, D. Fang, X. Chen, T. Xing, G. Ye, J. Zhang, Z. Wang, Exploiting dynamic scheduling for VM-based code obfuscation, Proceedings of the IEEE Trustcom/BigDataSE/ISPA, IEEE, 2016, pp. 489–496.
[304] H. Borck, M. Boddy, I.J.D. Silva, S. Harp, K. Hoyme, S. Johnston, A. Schwerdfeger, M. Southern, Frankencode: creating diverse programs using code clones, Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, Reengineering (SANER), 1 IEEE, 2016, pp. 604–608.
[305] P. Mäki, S. Rauti, S. Hosseinzadeh, L. Koivunen, V. Leppänen, Interface diversification in IoT operating systems, Proceedings of the IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC), ACM, 2016, pp. 304–309.
[306] P. Wang, S. Wang, J. Ming, Y. Jiang, D. Wu, Translingual obfuscation, Proceedings of the IEEE European Symposium on Security and Privacy (EuroS P), IEEE, 2016, pp. 128–144.
[307] Y. Peng, J. Liang, Q. Li, A control flow obfuscation method for android applications, Proceedings of the 4th International Conference on Cloud Computing and Intelligence Systems (CCIS), IEEE, 2016, pp. 94–98.
[308] L. Koivunen, S. Rauti, V. Leppänen, Applying internal interface diversification to IoT operating systems, Proceedings of the International Conference on Software Security and Assurance (ICSSA), IEEE, 2016, pp. 1–5.
[309] T.Y. Chen, F.C. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, Z.Q. Zhou, Metamorphic testing for cybersecurity, Comput. (Long Beach Calif) 49 (6) (2016) 48–55.
[310] W. Liu, W. Li, Unifying the method descriptor in Java obfuscation, Proceedings of the 2nd IEEE International Conference on Computer and Communications (ICCC), IEEE, 2016, pp. 1397–1401.
[311] R. Mohsen, A.M. Pinto, Evaluating obfuscation security: a quantitative approach, in: J. Garcia-Alfaro, E. Kranakis, G. Bonfante (Eds.), Foundations and Practice of Security, Springer International Publishing, Cham, 2016, pp. 174–192.
[312] M. Ceccato, P. Falcarin, A. Cabutto, Y.W. Frezghi, C.-A. Staicu, Search based clustering for protecting software with diversified updates, in: F. Sarro, K. Deb (Eds.), Search Based Software Engineering, Springer International Publishing, Cham, 2016, pp. 159–175.
[313] D. Xu, J. Ming, D. Wu, Generalized dynamic opaque predicates: A new control flow obfuscation method, in: M. Bishop, A.C.A. Nascimento (Eds.), Information Security, Springer International Publishing, Cham, 2016, pp. 323–342.
[314] J. Aycock, Obfuscation and Optimization, Springer International Publishing, Cham, pp. 173–203.
[315] G.-l. Cai, B.-s. Wang, W. Hu, T.-z. Wang, Moving target defense: state of the art and characteristics, Front. Inf. Technol. Electr. Eng. 17 (11) (2016) 1122–1153.
[316] A. Pawlowski, M. Contag, T. Holz, Probfuscation: an obfuscation approach using probabilistic control flows, in: J. Caballero, U. Zurutuza, R.J. Rodríguez (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Springer International Publishing, Cham, 2016, pp. 165–185.
[317] S. Rauti, S. Laurén, J. Uitto, S. Hosseinzadeh, J. Ruohonen, S. Hyrynsalmi,

V. Leppänen, A survey on internal interfaces used by exploits and implications on interface diversification, in: B.B. Brumley, J. Röning (Eds.), Secure IT Systems, Springer International Publishing, Cham, 2016, pp. 152–168.
[318] S. Pastrana, J. Tapiador, G. Suarez-Tangil, P. Peris-López, Avrand: a software-based defense against code reuse attacks for AVR embedded devices, in: J. Caballero, U. Zurutuza, R.J. Rodríguez (Eds.), Detection of Intrusions and Malware, and Vulnerability Assessment, Springer International Publishing, Cham, 2016, pp. 58–77.
[319] R. De Keulenaer, J. Maebe, K. De Bosschere, B. De Sutter, Link-time smart card code hardening, Int. J. Inf. Secur. 15 (2) (2016) 111–130.
[320] Y. Piao, J. Jung, J.H. Yi, Server-based code obfuscation scheme for apk tamper detection, Secur. Commun. Netw. 9 (6) (2016) 457–467.
[321] V. Balachandran, Sufatrio, D.J. Tan, V.L. Thing, Control flow obfuscation for android applications, Comput. Secur. 61 (2016) 72–93.
[322] S. Hosseinzadeh, S. Hyrynsalmi, V. Leppänen, Chapter 14 - obfuscation and diversification for securing the internet of things (IoT), in: R. Buyya, A.V. Dastjerdi (Eds.), Internet of Things, Morgan Kaufmann, 2016, pp. 259–274.
[323] S. Banescu, C. Lucaci, B. Krämer, A. Pretschner, VOT4CS: A virtualization obfuscation tool for C♯, Proceedings of the ACM Workshop on Software PROtection, SPRO '16, ACM, NY, USA, 2016, pp. 39–49.
[324] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, V. Leppänen, A survey on aims and environments of diversification and obfuscation in software security, in: B. Rachev, A. Smrikarov (Eds.), Proceedings of the 17th International Conference on Computer Systems and Technologies CompSysTech'16, ACM, 2016, pp. 113–120.
[325] R. Manikyam, J.T. McDonald, W.R. Mahoney, T.R. Andel, S.H. Russ, Comparing the effectiveness of commercial obfuscators against mate attacks, in: Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW '16, ACM, NY, USA, 2016, pp. 8:1–8:11.
[326] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, E. Weippl, Protecting software through obfuscation: can it keep pace with progress in code analysis?, ACM Comput. Surv. 49 (1) (2016) 4:1–4:37.
[327] J. Coffman, D.M. Kelly, C.C. Wellons, A.S. Gearhart, Rop gadget prevalence and survival under compiler-based binary diversification schemes, in: Proceedings of the ACM Workshop on Software PROtection, SPRO '16, ACM, NY, USA, 2016, pp. 15–26.
[328] B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, B. De Sutter, Tightly-coupled self-debugging software protection, in: Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW '16, ACM, NY, USA, 2016, pp. 7:1–7:10.
[329] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, A. Pretschner, Code obfuscation against symbolic execution attacks, in: Proceedings of the 32Nd Annual Conference on Computer Security Applications, ACSAC '16, ACM, NY, USA, 2016, pp. 189–200.
[330] J. Petke, Genetic improvement for code obfuscation, Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '16 Companion, ACM, NY, USA, 2016, pp. 1135–1136.
[331] M. Wu, Y. Zhang, X. Mi, Binary protection using dynamic fine-grained code hiding and obfuscation, in: Proceedings of the 4th International Conference on Information and Network Security, ICINS '16, ACM, NY, USA, 2016, pp. 1–8.
[332] H. Xu, Y. Zhou, M. Lyu, N-version obfuscation, in: Proceedings of the 2Nd ACM International Workshop on Cyber-Physical System Security, CPSS '16, ACM, NY, USA, 2016, pp. 22–33.
[333] S. Laurén, S. Rauti, V. Leppänen, An interface diversified honeypot for malware analysis, in: Proceedings of the 10th European Conference on Software Architecture Workshops, ECSAW '16, ACM, NY, USA, 2016, pp. 29:1–29:6.
[334] S. Blazy, A. Trieu, Formal verification of control-flow graph flattening, in: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP, ACM, NY, USA, 2016, pp. 176–187.
[335] H. Koo, M. Polychronakis, Juggling the gadgets: Binary-level code randomization using instruction displacement, in: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, ASIA CCS '16, ACM, NY, USA, 2016, pp. 23–34.
[336] K. Braden, L. Davi, C. Liebchen, A.R. Sadeghi, S. Crane, M. Franz, P. Larsen, Leakage-resilient layout randomization for mobile devices. Proceedings of the Network and Distributed System Security Symposium (NDSS), 2016 Internet Society, San Diego, CA, USA, 2016.
[337] K. Lu, W. Lee, S. Nürnberger, M. Backes, How to make ASLR win the clone wars: Runtime re-randomization. Proceedings of the NDSS, 2016 Internet Society, San Diego, CA, USA, 2016.
[338] G. Maisuradze, M. Backes, C. Rossow, What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses, Proceedings of the 25th USENIX Security Symposium, USENIX, Austin, TX, 2016, pp. 139–156.
[339] Y. Chen, Z. Wang, D. Whalley, L. Lu, Remix: on-demand live randomization, in: Proceedings of 6th ACM Conference on Data and Application Security and Privacy, CODASPY '16, ACM, NY, USA, 2016, pp. 50–61.
[340] M. Conti, S. Crane, T. Frassetto, A. Homescu, G. Koppen, P. Larsen, C. Liebchen, M. Perry, A.-R. Sadeghi, Selfrando: securing the tor browser against de-anonymization exploits, Proc. Priv. Enhancing Technol. 2016 (4) (2016) 454–469.
[341] S. Crane, A. Homescu, P. Larsen, Code randomization: haven't we solved this problem yet?, Proceedings of the IEEE Cybersecurity Development (SecDev), IEEE, 2016, pp. 124–129.
[342] D. Williams-King, G. Gobieski, K. Williams-King, J.P. Blake, X. Yuan, P. Colp, M. Zheng, V.P. Kemerlis, J. Yang, W. Aiello, Shuffler: fast and deployable continuous code re-randomization. Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16), (2016), pp. 367–382.
[343] J. Uitto, S. Rauti, V. Leppänen, Practical implications and requirements of diversifying interpreted languages, in: Proceedings of the 11th Annual Cyber and Information Security Research Conference, CISRC '16, ACM, NY, USA, 2016, pp. 14:1–14:4.
[344] N. Veeranna, B.C. Schafer, Efficient behavioral intellectual properties source code obfuscation for high-level synthesis, Proceedings of the 18th IEEE Latin American Test Symposium (LATS), IEEE, 2017, pp. 1–6.
[345] P. Kanani, K. Srivastava, J. Gandhi, D. Parekh, M. Gala, Obfuscation: maze of code, Proceedings of the 2nd International Conference on Communication Systems, Computing and IT Applications (CSCITA), IEEE, 2017, pp. 11–16.
[346] S. Wang, P. Wang, D. Wu, Composite software diversification, Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2017, pp. 284–294.
[347] H. Liu, C. Sun, Z. Su, Y. Jiang, M. Gu, J. Sun, Stochastic optimization of program obfuscation, Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE), IEEE Press, Piscataway, NJ, USA, 2017, pp. 221–231.
[348] T. Cho, H. Kim, J.H. Yi, Security assessment of code obfuscation based on dynamic monitoring in android things, IEEE Access 5 (2017) 6361–6371.
[349] M. Togan, A. Feraru, A. Popescu, Virtual machine for encrypted code execution, Proceedings of the 9th International Conference on Electronics, Computers and Artificial Intelligence (ECAI), IEEE, 2017, pp. 1–6.
[350] R. Tiella, M. Ceccato, Automatic generation of opaque constants based on the k-clique problem for resilient data obfuscation, Proceedings of the IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2017, pp. 182–192.
[351] Y. Peng, G. Su, B. Tian, M. Sun, Q. Li, Control flow obfuscation based protection method for android applications, China Commun. 14 (11) (2017) 247–259.
[352] Z. Li, X.Y. Jing, X. Zhu, H. Zhang, B. Xu, S. Ying, On the multiple sources and privacy preservation issues for heterogeneous defect prediction, IEEE Trans. Softw. Eng. PP (99) (2017) 1–21.
[353] X. Chen, H. Bos, C. Giuffrida, Codearmor: virtualizing the code space to counter disclosure attacks, Proceedings of the 2017 IEEE European Symposium on Security and Privacy (EuroS P), IEEE, 2017, pp. 514–529.
[354] D. Canavese, L. Regano, C. Basile, A. Viticchié, Estimating software obfuscation potency with artificial neural networks, in: G. Livraga, C. Mitchell (Eds.), Security and Trust Management, Springer International Publishing, Cham, 2017, pp. 193–202.
[355] B. Zhao, Z. Tang, Z. Li, L. Song, X. Gong, D. Fang, F. Liu, Z. Wang, Dexpro: a bytecode level code protection system for android applications, in: S. Wen, W. Wu, A. Castiglione (Eds.), Cyberspace Safety and Security, Springer International Publishing, Cham, 2017, pp. 367–382.
[356] S. Hosseinzadeh, S. Laurén, S. Rauti, S. Hyrynsalmi, M. Conti, V. Leppänen, Obfuscation and diversification for securing cloud computing, in: V. Chang, M. Ramachandran, R.J. Walters, G. Wills (Eds.), Enterprise Security, Springer International Publishing, Cham, 2017, pp. 179–202.
[357] X. Tang, Y. Liang, X. Ma, Y. Lin, D. Gao, On the effectiveness of code-reuse-based android application obfuscation, in: S. Hong, J.H. Park (Eds.), Information Security and Cryptology – ICISC 2016, Springer International Publishing, Cham, 2017, pp. 333–349.
[358] R. Géraud, M. Koscina, P. Lenczner, D. Naccache, D. Saulpic, Generating functionally equivalent programs having non-isomorphic control-flow graphs, in: H. Lipmaa, A. Mitrokotsa, R. Matulevičius (Eds.), Secure IT Systems, Springer International Publishing, Cham, 2017, pp. 265–279.
[359] M. Morton, H. Koo, F. Li, K.Z. Snow, M. Polychronakis, F. Monrose, Defeating zombie gadgets by re-randomizing code upon disclosure, in: E. Bodden, M. Payer, E. Athanasopoulos (Eds.), Engineering Secure Software and Systems, Springer International Publishing, Cham, 2017, pp. 143–160.
[360] W. Holder, J.T. McDonald, T.R. Andel, Evaluating optimal phase ordering in obfuscation executives, in: Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop, SSPREW-7, ACM, NY, USA, 2017, pp. 6:1–6:12.
[361] B. Johansson, P. Lantz, M. Liljenstam, Lightweight dispatcher constructions for control flow flattening, in: Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop, SSPREW-7, ACM, NY, USA, 2017, pp. 2:1–2:12.
[362] K. Lim, J. Jeong, S.-j. Cho, J. Choi, M. Park, S. Han, S. Jhang, An anti-reverse engineering technique using native code and obfuscator-llvm for android applications, in: Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS '17, ACM, NY, USA, 2017, pp. 217–221.
[363] R.N. Ismanto, M. Salman, Improving security level through obfuscation technique for source code protection using AES algorithm, in: Proceedings of 7th International Conference on Communication and Network Security, ICCNS 2017, ACM, NY, USA, 2017, pp. 18–22.
[364] M. Zhang, M. Polychronakis, R. Sekar, Protecting COTS binaries from disclosure-guided code reuse attacks, in: Proceedings of the 33rd Annual Computer Security Applications Conference, ACSAC, ACM, NY, USA, 2017, pp. 128–140.
[365] M. Pomonis, T. Petsios, A.D. Keromytis, M. Polychronakis, V.P. Kemerlis, kR^X: Comprehensive kernel protection against just-in-time code reuse, in: Proceedings of the Twelfth European Conference on Computer Systems, EuroSys '17, ACM, NY, USA, 2017, pp. 420–436.
[366] S. Halevi, T. Halevi, V. Shoup, N. Stephens-Davidowitz, Implementing BP-obfuscation using graph-induced encoding, in: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS '17, ACM, NY, USA, 2017, pp. 783–798.
[367] T. Groß, T. Müller, Protecting JavaScript apps from code analysis, in: Proceedings of the 4th Workshop on Security in Highly Connected IT Systems, SHCIS '17, ACM, NY, USA, 2017, pp. 1–6.

Publication II

Obfuscation and Diversification for Securing Cloud Computing

Shohreh Hosseinzadeh, Samuel Laurén, Sampsa Rauti, Sami Hyrynsalmi, Mauro Conti, Ville Leppänen. In: Victor Chang, Muthu Ramachandran, Robert J. Walters, Gary Wills (Eds.), Enterprise Security, Lecture Notes in Computer Science 10131, 179–202. Springer, 2017.

Obfuscation and Diversification for Securing Cloud Computing

Shohreh Hosseinzadeh[1(✉)], Samuel Laurén[1], Sampsa Rauti[1], Sami Hyrynsalmi[1], Mauro Conti[2], and Ville Leppänen[1]

[1] Department of Future Technologies, University of Turku, Agora 4th Floor, Vesilinnantie 5, 20500 Turku, Finland
{shohos, smrlau, sjprau, sthyry, ville.leppanen}@utu.fi
[2] Department of Mathematics, University of Padua, Padua, Italy
conti@math.unipd.it

Abstract. The evolution of cloud computing and advancement of its services has motivated the organizations and enterprises to move towards the cloud, in order to provide their services to their customers, with greater ease and higher efficiency. Utilizing the cloud-based services, on one hand has brought along numerous compelling benefits and, on the other hand, has raised concerns regarding the security and privacy of the data on the cloud, which is still an ongoing challenge. In this regard, there has been a large body of research on improving the security and privacy in cloud computing. In this chapter, we first study the status of security and privacy in cloud computing. Then among all the existing security techniques, we narrow our focus on *obfuscation* and *diversification* techniques. We present the state-of-the-art review in this field of study, how these two techniques have been used in cloud computing to improve security. Finally, we propose an approach that uses these two techniques with the aim of improving the security in cloud computing environment and preserve the privacy of its users.

Keywords: Cloud computing · Enterprise security · Security · Privacy · Obfuscation · Diversification

1 Introduction

The recent changes in the business world have made the organizations and enterprises more interested in using cloud to share their services and resources remotely to their users. For this purpose, cloud computing offers three different models (Mell and Grance 2011): Software as a service (SaaS), Platform as a service (PaaS), and Infrastructure as a service (IaaS). In IaaS model the services that are offered by the service provider include computing resources, storage, and virtual machines. The PaaS model presents computing platforms to the business and its end users. In SaaS, the main services offered by the service providers are the applications that are hosted and executed on the cloud and are available to the customers through the network, typically over the Internet. Depending on the need of the enterprise, a suitable delivery model is deployed.

V. Chang et al. (Eds.): Enterprise Security, LNCS 10131, pp. 179–202, 2017.
DOI: 10.1007/978-3-319-54380-2_8

The advancements in the cloud computing have facilitated the business, organizations, and enterprises with services providing lower cost and higher performance, scalability, and availability. Due to these advantages, cloud computing has become a highly demanded technology and organizations are relying on cloud services more and more. However, by using the cloud, more data is stored outside the organization's perimeters, which raises concerns about the security and privacy of the data. Therefore, it is significant for the cloud service providers to employ effective practices to secure the cloud computing infrastructure and preserve the privacy of its users. In the context of the cloud computing security, there exist many different measures that protect the cloud infrastructure. Some of these measures consider the cloud as an untrusted or malicious infrastructure that the user's data should be protected from (e.g., the cloud uses the data without the user's consent). While some other measures protect the infrastructure from the external intrusions.

Obfuscation and diversification are two propitious software security techniques that have been employed in various domains, mainly to impede malware (i.e., malicious software) (Skoudis 2004). These techniques have also been used to provide security in cloud computing as well. In a previous work[1], we have systematically surveyed the studies that use obfuscation and diversification techniques with the aim of improving the security in cloud computing environment (see the details of this study in Sect. 3.1). By analyzing the collected data, we managed to identify the areas that have gained more attention by the previous research, and also the areas that have remained intact and potential for further research. The results of the survey motivated us to propose a diversification approach, aiming at improving the security in cloud computing. We demonstrate this approach by applying obfuscation on client-side JavaScript components of an application. As such, we make it complicated for a piece of malware to gain knowledge about the internal structure of the application and perform its malicious attack. Moreover, we distribute unique versions of the application to the computers, which in the end mitigates the risk of massive-scale attacks (more detail in Sect. 4).

This book chapter is structured as follows: Sect. 2 discusses the security and privacy in domain of cloud computing, available security threats, and different aspects of security, concerning the cloud computing technology. In Sect. 3 we introduce the terms and techniques that are used in the proposed approach, and we present the state of this field of study, i.e., how these techniques have previously been used to boost the security of cloud computing. In Sect. 4, we present our proposed approach in detail. Conclusions come in Sect. 5.

2 Security and Privacy in Cloud Computing

Cloud computing is an evolving technology with new capabilities and services that have remarkable benefits when compared to more traditional service providing approaches. The services are delivered with lower cost (in usage as it is pay-for-use, in

[1] This book chapter is a re-written extended version of our previous study (Hosseinzadeh et al. 2015).

disaster recovery, in data storage solutions), greater ease, less complexity, higher availability and scalability, and also faster deployment. These compelling benefits have motivated the enterprises to adopt cloud solutions in their architectures and deliver their services over cloud. Depending on the need of the enterprise and how large the organization is, different deployment models are available, including public clouds, private clouds, community cloud, and hybrid clouds (Mell and Grance 2011; Mather et al. 2009). Again, depending on the need of the enterprise, cloud providers offer three different delivery models, i.e., SaaS, IaaS, and PaaS. In the following we discuss various deployment models, and various business models in cloud computing (CSA 2016).

In a *public cloud* (or external cloud), a third party vendor is responsible for hosting, operating, and managing the cloud. A common infrastructure is used to serve multiple customers, which means that the customers are not required to acquire for any software, hardware, and network devices. This makes the public cloud a suitable model for the enterprises that wish to invest less and manage the costs efficiently. The security in a public cloud is managed by the third party, which leaves less control for the organization and its users over the security (Rhoton et al. 2013). A kind of opposite solution to that is *private cloud* (or internal cloud), where the organization's customers are in charge of managing the cloud. The storage, computing, and network are dedicated to the organization owning the cloud, and not shared with other organizations. This enables the customers to have a higher control on security management and have more insight about logical and physical aspects of the cloud infrastructure. *Community cloud* refers to the type of clouds that are used exclusively by a community of customers from enterprises with common requirements and concerns (e.g., policies, security requirements, and compliance considerations). The last model is *hybrid cloud* that is the composition of several clouds (private, public, and community). According to the needs and budget of the enterprise and how critical its resources are, a suitable deployment model is chosen that can serve the enterprise's needs the best. For instance, an enterprise pays more for private cloud and has better security control over its shared resources, while it spends less on a public cloud for which it has less security control (CSA 2016; Mather et al. 2009).

As mentioned before, cloud service providers use three different business models to deliver their services to the end users: IaaS, PaaS, and SaaS. IaaS is the foundation of the cloud, PaaS comes on top of that, and SaaS is built upon PaaS. Each of these delivery models has different security issues, and is prone to different types of security threats, and therefore, it requires different levels of security.

The IaaS service model offers capabilities such as storage, processing, networks, and computing resources to the consumers. The end user does not manage the underlying infrastructure of the cloud, but has control over applications, operating system and storage. IaaS has made it a lot easier for the enterprises to deliver their services, in a way that they no longer worry about provisioning and managing the infrastructure and dealing with the underlying complexities. In addition to that, it has made it cheaper for businesses, in a way that instead of paying for the data centers and hosting companies, they only need to spend for the resources they consume to IaaS providers (Mell and Grance 2011; Mather et al. 2009).

The PaaS model is built upon IaaS and offers a development environment to the developers to develop their applications, without worrying about the underlying infrastructure. The offered services consist of a complete set of software development kit, ranging from design to testing and maintenance. The consumers of this model do not have control on the beneath services (e.g. the operating system, server, network, and the storage), but they can manage the application-hosting environment (Mell and Grance 2011; Mather et al. 2009). The dark side of these advantages is that the PaaS infrastructure can also be used by a hacker to malicious purposes (e.g., running the malware codes and commands).

In the SaaS model, the service providers host the applications remotely and make them available to the users, when requested, over the Internet. SaaS is an advantageous model for the IT enterprises and their customers, as it is more cost-effective and has better operational efficiency. However, there are still concerns about the security of the data store and software, which the vendors are required to address them (Mell and Grance 2011; Mather et al. 2009).

In addition to these three main service delivery models, the cloud offers other models and the infrastructure is utilized these days for many other purposes, such as Security-as-a-Service (SECaaS). Using this service model, many security vendors deliver their security solutions using cloud services. That is to say, the security management services are outsourced to an external service provider, and delivered to the users over the Internet. SECaaS applications can be in the form of anti-virus, anti-spam, and malware detection programs. The programs operate on the cloud, instead of client-side installed software, and with no need for on-premises hardware (Varadharajan and Tupakula 2014).

Employing cloud services by an enterprise is a two-edged sword, meaning that it has both positive and negative impacts. On one hand, by outsourcing and shifting some responsibilities from the enterprise to the cloud, fewer unwanted incidents are expected to occur. This is due to the fact that the cloud providers have a more advanced and experienced position in offering more secure services that are supported by their specialized staff, and also there are incident management plans for the case of break outs. However, this transferring of the responsibilities, on the other hand, decreases the control of the enterprise over the critical services. In addition to that, storing the data outside the organization's firewalls raises concerns about potential vulnerabilities and possible leaks. For instance, if the information fall into wrong hands (e.g., exposed to hackers or competitors), it results in loss of customer's confidence, damage to the organization's reputation, and even legal and financial penalties for the organization. On this basis, enterprises that are planning to adopt cloud services put together the positive and negative impacts, weigh them up, and do risk assessment (Rhoton et al. 2013).

In spite of the benefits of adopting cloud-based services in the enterprises, there exist still some barriers. Among all, security and privacy are the most significant barriers. The fundamental challenges related to the cloud security are the security of the data storage, security in data transmission, application security, and security of the third party resources (Subashini and Kavitha 2011). Among all the security risks associated with the cloud, the followings are the top severe security threats reported by the Cloud Security Alliance (CSA) (Top Threats Working Group 2013):

- Data breach: As a result of a malicious intrusion the (sensitive) data may be disclosed to unwanted parties, including the attacker and competitors.
- Data loss: The data stored on the cloud could be lost due to an attack, unintentional/accidental deletion of the data by the service provider, and physical corruption of the system infrastructure.
- Hijacking of the accounts and traffic: Attackers by getting access to the users' credentials, through phishing and exploiting the software vulnerabilities can read or alter the users' activities. This consequently puts the confidentiality, integrity, and availability of the system at risk.
- Insecure Application Program Interfaces (API): Clients use the SW interfaces to interact with the cloud. These APIs should be sufficiently secure to protect both the consumer and the service provider.
- Denial of service: An intruder by sending illegitimate requests to the service provider attempts to occupy the resources, so to disable/slow down the cloud to process the legitimate requests.
- Malicious insider: The adversary is not always an outsider, but can be a person inside the cloud system who has an authorized access to the data and intentionally misuses such authorization.
- Abuse of the cloud service: The cloud computing serves the organizations with extensive computational power; however, this power could be misused by a malicious user to perform his belligerent action.
- Insufficient due diligence: Before the organizations move their services to the cloud, it is significant to have a proper understanding of the capabilities and adaptabilities of their resources with the cloud technologies.
- Shared technology issues: Sharing platforms, infrastructures, and applications has made the delivery of the cloud services feasible; however, such sharing has the drawback that vulnerability in a single piece of shared component can be propagated potentially to the entire cloud.

In securing the cloud computing environment various aspects should be taken into account. The International Information Systems Security Certification Consortium $(ISC)^2$ (ISC 2016) has presented taxonomy of the security domains concerning the cloud computing, which covers the following aspects: physical security, access control, telecommunications and network security, cryptography, application security, operation security, information security and risk management practices, and business continuity and disaster recovery planning.

Many different approaches have been proposed in the literature for overcoming these security problems, for instance multi-layered security and large scale penetration testing (Chang 2015; Chang et al. 2016; Chang and Ramachandran 2016).

Security and privacy come hand in hand, in other words, a more secure system better protects the privacy of its users. Therefore, while integrating cloud in organization's architecture, it is highly significant for the enterprise to assure that a cloud service provider is considering all the security aspects, and adequately addressing the privacy regulations.

Considering the fact that the proposed approach in this study is aimed at securing the application, and protecting SaaS and PaaS models, in the following section we study the state of cloud application security.

2.1 Application Security in Cloud Computing

Talking about the IaaS model, it is more straightforward to provide protection by hardening the platform through allowing the traffic from trusted IP addresses, running anti-virus programs, applying security patches, and so on. However, when it comes to PaaS and SaaS models, this may not be the case; since in these models, ensuring the security of the platform is the service provider's responsibility (Rhoton et al. 2013). Moreover, in SaaS, there is less transparency and visibility about how the data is stored and secured. This makes it more difficult for the enterprises to trust the service provider.

Application security covers the measures and practices taken throughout the software development life-cycle to reduce the vulnerabilities and flaws. Because of the fact that the cloud-base applications are connected directly to the Internet, cloud offers less physical security compared to traditional data centers and service providers. Also because of the co-mingled data and multi-tenancy behavior of the cloud, the cloud's applications are prone to additional attack vectors.

Recent security incidents clearly show that the exploits by taking advantage of the software flaws and vulnerabilities, make the web applications the leading targets for attacks. Web applications are the simplest and the most profitable targets, from the attackers' perspective. Especially, in the case of cloud computing that the applications are accessed through the user's browser, website security is the sole means to impede the attacks. Moreover, the security breaches through exploiting the applications and web services have shown to be pretty severe and have led to big losses. Stuxnet (Chen and Abu-Nimeh 2011) is one example of infecting the software with the aim of affecting critical physical infrastructures and industrial control systems. The other example is using SQL injection to steal the debit/credit card numbers, which in the end resulted in 1 million withdrawals from the ATM machines worldwide (CSA 2016).

In spite of the significance of the application security, it has been considered as an afterthought in many enterprises. In other words, application security has seldom been the top priority and the main focus for neither the security practitioners and nor the enterprises and less security budget has been allocated on it (CSA 2016).

On this basis, more consideration is required both from the business side by shifting more budget to application security and also from the security team to concentrate more on securing the web applications, the most exposed component of the business.

As in other domains, application security in cloud computing is a crucial component in operational IT strategy. Regardless of where an application is residing, the enterprise is responsible for ensuring the effectiveness of the security practices to

protect the application. Also, as we discussed earlier, the nature of cloud computing environment introduces additional risks compared to on-premise applications and web services.

3 Obfuscation and Diversification for Securing Cloud Computing

Code obfuscation refers to the deliberate act of scrambling the program's code and transforming it in a way that it becomes harder to read (Collberg et al. 1997). This new version of the code is functionally similar to the original code, while syntactically different. This means that even though the obfuscated code has different implementation, given the same input, it produces the same output. The main purpose of code obfuscation is to make the understanding of code and its functionality more complicated and to prevent the act of malicious reverse engineering.

Figure 1 is an example of obfuscated code that clearly shows how much harder it can become to read and comprehend the code after it is obfuscated. With no doubt, within a given time an attacker may succeed in reverse engineering the obfuscated code and breaking it: however, it is harder and costlier now, compared to the original code.

a)

```
function setText(data) {
  document.getElementById("myDiv").innerHTML = data;
}
```

b)

```
function ghds3x(n) {
  h = "\x69\u006En\u0065r\x48T\u004DL";
  a="s c v o v d h e , n i";x=a.split(" ");b="gztxleWentBsyf";
  r=b.replace("z",x[7]).replace("x","E").replace("s","").replace("f","I")
  ["repl" + "ace"]("W","m")+"d"; c="my"+String.fromCharCode(68)+x[10]+"v";
  s=x[5]+x[3]+x[1]+"um"+x[7]+x[9]+"t";d=this[s][r](c);if(+!![])
  { d[h]=n; } else { d[h]=c; }
}
```

Fig. 1. a) a piece of JavaScript code, and b) an obfuscated version of the same code

In the literature, many different obfuscation mechanisms have been proposed (Popov et al. 2007; Linn and Debray 2003). Each of these mechanisms targets various parts of the code to apply the obfuscation transformation. Among all, the techniques that attempt to obfuscate the control of the program are the most commonly used (Nagra and Collberg 2009). These techniques alter the control flow of the program, or generate a fake one, so it would be more challenging for a malicious analyzer to understand the code. To this end, bogus insertion (Drape and Majumdar 2007), and opaque predicates (Collberg et al. 1998) are effective control flow obfuscation techniques.

Software diversification aims at generating unique instances of software in a way that they appear with different syntax but equivalent functionality (Cohen 1993). Diversification breaks the idea of developing and distributing the software in a monoculture manner, and introduces multiculturalism to software design. In the other words, the identical designs of the software instances make them have similar vulnerabilities and are prone to similar types of security threats. This offers the opportunity to an attacker to design an attack model to exploit those vulnerabilities and easily compromise a wide number of execution platforms (e.g., computers). The risk of this kind of massive-scale attacks can be mitigated through diversifying the software versions, so that the same attack model will not be effectual on all instances. The way a program is diversified is kept secret and pieces of malware that do not know the secret cannot interact with the environment and eventually become ineffective. However, the created secret has to be propagated to the trusted applications, so it will still be feasible for them to access the resources. In the worst case scenario, even if the attacker gains the secret of diversification of one instance, that secret is specified to that computer and a costly analysis is required to find out other secrets to attack other computers. There have been survey studies surveying software diversity (Larsen et al. 2014; Baudry and Monperrus 2015).

A particular version of diversification is *interface diversification* which is applied to internal interfaces of software (APIs or instruction sets of languages). For example, the system call interface (for accessing all kinds of resources of a system) is one typical *internal* interface which can be changed without sharing the details of new internal interface to external parties (e.g. malware) (Rauti et al. 2014). Of course, the details on diversified internal interface need to be propagated to all legal applications so that those programs can still use the system's resources (Lauren et al. 2014).

Figure 2 illustrates distribution of diversified versions of program P among the users. Each of the programs P1, P2, and P3 are unique in structure and diversified differently. Thus, one single attack model does not work for multiple systems, and

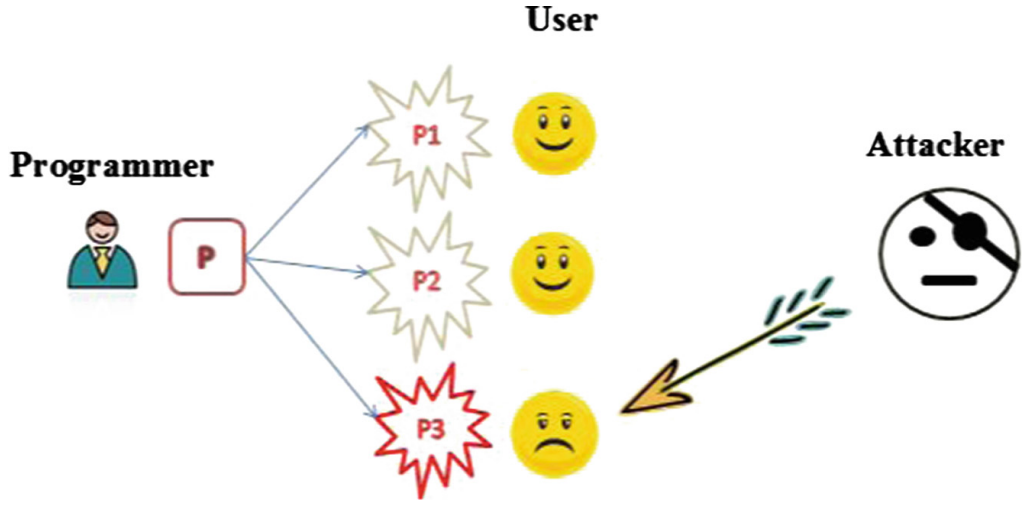


Fig. 2. Diversification generates unique versions of software. Therefore, even if one copy of software is breached, other copies are safe.

attack models need to be designed to be system-specific. When program P3 is attacked, other versions are still safe.

Program bugs left at the development time are inevitable and cause software vulnerabilities. Some of these vulnerabilities are not known, while releasing the software. Later on, a malicious person can gain knowledge about the system and its vulnerabilities, and write a piece of malware to exploit those vulnerabilities performing a successful attack. Especially, the interface diversification techniques can be helpful in preventing such zero day type of attacks, since the malicious person no longer automatically know the necessary (for malware) internal interfaces for accessing resources.

In general, diversification and obfuscation techniques do not attempt to remove these vulnerabilities, but attempt to prevent (or make it hard, at least) the attacker/malware to taking advantage of them to run its malicious code. Obfuscating and diversifying the internal interfaces of the system makes it challenging for malware to attain the required knowledge about the system, how to call the systems interfaces, in order to execute its malicious code.

3.1 Related Work on Security of Cloud Through Obfuscation and Diversification

As mentioned before, diversification and obfuscation techniques have been used in different domains to provide security, including cloud computing. In a previous study (Hosseinzadeh et al. 2015), we systematically studies in what ways these two techniques have been used in cloud computing environment with the aim of improving the security. As the result of the search, we collected 43 studies that were discussing diversification and obfuscation as the techniques for improving the security in the cloud and protect the privacy of its users, and we classified them based on how the techniques are used to this aim. After extracting data from those studies, we identified that the obfuscation and diversification techniques are used in nine different ways to boost the security and privacy of the cloud, including: (1) generating noise obfuscation, (2) client-side data obfuscation as a middleware, (3) general data obfuscation, (4) source code obfuscation, (5) location obfuscation, (6) file splitting and storing on separate clouds, (7) encryption as obfuscation, (8) diversification, and (9) cloud security by virtue of securing the browser. Figure 3 illustrates these categories with the number of studies in each group.

Many of the cloud service providers are complying with the policies and regulations in order to protect the privacy of their customers. However, there exist a wide number of service providers that may record the collected data from the customers, deduce and misuse the private information without user's consent. Hence, there is a need for practices to be taken at client side (without service provider's interference) to protect the privacy. Obfuscation and diversification techniques were employed to protect the data "from the cloud". In majority of studied works, the cloud service provider was considered as untrusted/malicious.

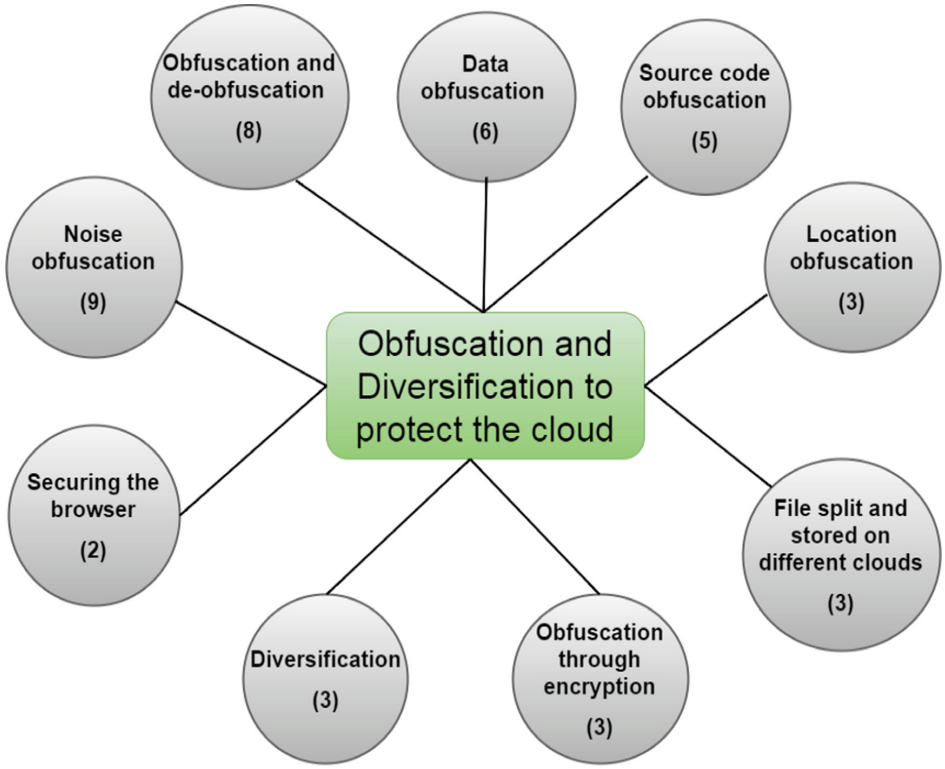


Fig. 3. The related studies on security and privacy in cloud computing through obfuscation and diversification techniques.

In the following, we explain the nine different ways that obfuscation/diversification have been used in the literature for protecting the cloud from security threats, and also protecting the user's privacy from the malicious cloud.

- Generating noise obfuscation: This approach resides on the client side and tries confuses the malicious cloud by injecting irrelevant requests that are similar to legitimate requests of the user (called noise) into the user's service requests, i.e., the requests sent from the customer to the cloud. In this way, the occurrence probability of the legitimate requests and the noise requests become the same, so it becomes difficult for the cloud to distinguish the real request. Noise generation strategy, conceals real requests coming from the users, and therefore, lessens the probability of request being revealed. As the result, the privacy of the customer is protected.
- Client-side data obfuscation as a middleware: This method protects the data from the untrusted service provider while the data is stored or processed on the cloud. A privacy managing middleware on the client side obfuscates the (sensitive) data using a secret key which is chosen and kept by the user. The obfuscated data is sent to the cloud and is processed on the cloud without being de-obfuscated. This is because the key is kept secret to the user and the cloud does not have the key to de-obfuscate the data. The result of the process is sent to the user and is de-obfuscated on the client side, so the user sees the plain data.

- General data obfuscation: In this class obfuscation, some transformations that are made into the user's data, which make them harder to read/understood. This method can be used to protect the user's identity information, the data stored on the database of the cloud, and the user's behavioral pattern. Data obfuscation makes the user's confidential harder to be exposed, and therefore, protects the user's data privacy.
- Source code obfuscation: As explained before, source code obfuscation is a technique for protecting the software from reverse engineering. This method can also be used for securing the cloud's software from attacks and risk of malware.
- Location obfuscation: As we know, there exist services that rely on the physical information of the user to provide the services. This includes privacy concerns on revealing the precise location of the user (e.g., concerns about locating and tracking down the user). To this end, obfuscating the location information is a technique to make the exact location imprecise through generalizing, or slightly altering the precise location to avoid the actual position being exposed and consequently, preserve the location privacy.
- File splitting and storing on separate clouds: The idea in this obfuscation strategy is to divide the data/files into different sectors and store them on different clouds. This approach ensures not only the security, but also the availability of the data. If one cloud is attacked, only one part will be leaked, and the other parts are safe.
- Encryption as obfuscation: Obfuscation could be attained through cryptographic techniques. For instance, homomorphic encryption and one-way hash function are examples of this obfuscation strategy. Obfuscating the data reduces the risk of data leakage and even if the data is leaked, it is quite useless, as it is scrambled. Therefore, it is a beneficial technique in preserving the confidentiality of the data on the cloud.
- Diversification: As discussed before, different components could be the target for diversification depending on the security need of the system. In cloud computing paradigm, it is proposed to continuously diversify the execution environment, so to shorten the time for the attacker to learn the execution environment and the vulnerabilities of it. The execution environment changes to a new environment, before the attacker gets the chance to obtain sufficient knowledge about it.
- Cloud security through securing the browser: In this idea a plug-in is embedded in the user's web browser, which has the capability of data obfuscation and hybrid authentication. Therein, the security and the privacy of the data are addressed in the web browser.

Table 1 lists all the papers that are discussing diversification and obfuscation as the promising security techniques in cloud computing environment. Based on how the studies use diversification/obfuscation is in cloud computing, they fall into nine different categories.

Table 1. List of the studies

No.	Description of method
Category 1: Generating noise obfuscation	
1	(Zhang et al. 2013): Injecting noise requests in user's request makes it difficult for the cloud to distinguish the legitimate request. The paper considers noise obfuscation as a way for privacy-leakage-tolerance
2	(Yang et al. 2013): This paper proposes noise generation approach as a way to obfuscate the data while the characteristic of the data is not changed. The main goal is to protect the privacy in the domain of data statics and data mining
3	(Zhang et al. 2012b): In this paper, Time-series Pattern Strategy Noise Generation (TPNGS) is used to create a pattern based on the previous requests that the user has made, and with the help of this pattern predict the occurrence probability of the future requests. This approach makes the real requests of the user vague, and protects the privacy of the client from a malicious cloud
4	(Zhang et al. 2015): In this work, noise obfuscation approach considers occurrence probability fluctuation as a way to disguise the customer's data
5	(Zhang et al. 2012c): Noise injection is discussed in this paper as a method to confuse the malicious cloud provider, with the aim of privacy protection
6	(Zhang et al. 2012a): Injecting noise (= irrelevant requests) into the user's request makes the occurrence possibility of the real and the noise requests the same, and thus make them indistinguishable
7	(Lamanna et al. 2012): This paper considers homomorphic encryption, oblivious transfer, and query obfuscation in the proxy as the techniques to protect the information from an untrusted cloud. Query obfuscation aims at generating random noisy/fake queries and confusing for the cloud
8	(Zhang et al. 2012d): Noise obfuscation disguises the occurrence probability of the user's requests. In this way, the user's personal information is kept safe, and therefore, the privacy is conserved
9	(Liu et al. 2012): In this paper, generating noise in user's requests is discussed as a way to protect the privacy
Category 2: Client-side data obfuscation as a middleware	
10	(Arockiam and Monikandan 2014): Before sending the data to the cloud, encryption and obfuscation techniques are used to ensure the confidentiality of the data. Obfuscation is used for the numerical data types, while encryption is applied on alphabetical type of data
11	(Tian et al. 2011): This paper suggests that the user's information be encrypted before being sent to the cloud. This encrypted data is decrypted only on the client side
12	(Yau and An 2010): The proposed approach protects the customer's data from malicious cloud through data obfuscation, information hiding, and separating the software and the infrastructure of the service provider
13	(Mowbray et al. 2012): This paper introduces a privacy manager that protects the user's private information by obfuscating them before delivering to the cloud. The key used for this purpose is selected by the privacy manager. The same key is used to de-obfuscate the processed data received from the cloud. They use the term obfuscation rather than encryption, since the data is partially obfuscated and some parts remain intact
14	(Pearson et al. 2009): This work presents a mathematical formulation for obfuscation, and also a privacy manager founded on obfuscation and de-obfuscation approaches

(*continued*)

Table 1. (*continued*)

No.	Description of method
15	(Mowbray and Pearson 2009): This paper proposes a privacy managing technique based on obfuscation and de-obfuscation approaches, to control the data transferred to the cloud. User's information is obfuscated using the key selected by user, and then sent to the cloud. This key is kept secret by the user, so the cloud is never able to de-obfuscate the data
16	(Govinda and Sathiyamoorthy 2012): In this approach customer's confidential data is obfuscated before being sent to the cloud service provider. The result of the processed data is sent back to the customer. There, the data is de-obfuscated on the client side using the user's secret key
17	(Patibandla et al. 2012): In this work, a privacy manager software is presented that obfuscates the user's sensitive data, prior to send to the cloud, based on the user's preferences
Category 3: General data obfuscation	
18	(Reiss et al. 2012): This paper proposes a systematic obfuscation approach that aims at protecting personal data. The obfuscation techniques used are: (a) transforming: changing the information into another format, (b) sub-setting: selecting a particular fraction of data, (c) culling: deleting a particular fraction of data, and (d) aggregation
19	(Kuzu et al. 2014): Data obfuscation is an advantageous solution to protect the data that is stored in the cloud's database. Besides, the access patterns could be obfuscated and protected as well
20	(Vleju 2012): The confidential information of the user can be protected through obfuscation. For instance, obfuscating the identification information conceals the user's real identity. After obfuscation is applied, the data can be deciphered only by the user
21	(Li et al. 2011): To protect the data privacy in SaaS, data obfuscation is proposed as a beneficial technique
22	(Qin et al. 2014): This paper proposes an algorithm based on obfuscation techniques to protect the confidential information that exist in CNF (Conjunctive Normal Form) format
23	(Tapiador et al. 2012): Typically, a user's decisions and behavior follow a similar pattern. Analyzing this pattern helps in foreseeing the future behavior which raises privacy concerns. Obfuscating the user's behavioral pattern, make this information inaccessible, or at least makes it harder to access
24	(Kansal et al. 2015): This paper proposes image obfuscation as a technique to hide and obfuscate an image (for instance by hiding the position of the pixels or the colors). For this purpose, the paper integrates the compression and secret sharing to produce multiple numbers of shadow images
Category 4: Source code obfuscation	
25	(Bertholon et al. 2013a): JavaScript is the language that is widely used in today's web services. To protect the JavaScript code, obfuscation is proposed to make it harder to reverse engineer
26	(Bertholon et al. 2013b): The paper presents a framework that transforms/obfuscates the source code of the C program into a jumbled form
27	(Hataba and El-Mahdy 2012): This paper is a survey of existing obfuscation techniques that aim at making the reverse engineering harder

(*continued*)

Table 1. (*continued*)

No.	Description of method
28	(Bertholon et al. 2014): JSHADOF framework is designed to obfuscate the JavaScript code. The target of the transformation is the source code in cloud computing web services
29	(Omar et al. 2014): This paper uses control-flow obfuscation and junk code insertion to present a threat-based obfuscation technique
Category 5: Location obfuscation	
30	(Karuppanan et al. 2012): This paper states that the user's private information needs to be protected from being disclosed. Location obfuscation is proposed to conceal the user's location
31	(Skvortsov et al. 2012): The paper states that the Location Services (LS) present services based on the location information of the user, which brings along privacy concerns. Location obfuscation solves this problem by making this information appear imprecise
32	(Agir et al. 2014): This paper considers location obfuscation as a way to confuse the server about the location of the user
Category 6: File splitting and storing on separate clouds	
33	(Celesti et al. 2014): In this work, data obfuscation is done through dividing the files and storing them on multiple clouds. In this way, each cloud has partial view to the file
34	(Ryan and Falvey 2012): In order to obfuscate the data, this work proposes splitting the data and storing them on geographically separated data stores
35	(Villari et al. 2013): To keep the data confidential, it is proposed to spread the data over various clouds
Category 7: Encryption as obfuscation	
36	(Padilha and Pedone 2015): The most common way to achieve obfuscation is to employ cryptographic approaches. Secret sharing is one practical example, in this regard
37	(Gao-xiang et al. 2013): This paper proposes achieving the obfuscation through homomorphic encryption
38	(Furukawa et al. 2013): This paper studies point function obfuscation which relies on one way hash functions
Category 8: Diversification	
39	(Tunc et al. 2014): Moving target defenses aim at continuously altering the execution environment of the system and its configurations, in order to make it challenging and costly for the intruder to learn about the environment and discover its vulnerabilities. This paper proposes diversification of the cloud's execution environment
40	(Yang et al. 2014): This paper proposes to continuously change the execution environment and also the platforms used to execute them. Hence, till the time that the attacker learns the execution environment, it has changed. Moreover, hardware redundancy is introduced as a way to increase the tolerance to the attacks
41	(Guo and Bhattacharya 2014): Design diversity is proposed in this paper for the cloud infrastructure. The target of the diversification is the configuration of virtual replicas. This increases the resiliency of the service, in case of possible attacks
Category 9: Cloud security through securing the browser	
42	(Prasadreddy et al. 2011): This paper proposes a plug-in for the user's web browser that offers double authentication and hybrid obfuscation for the data, and protects the security and privacy of the cloud in this way
43	(Palanques et al. 2012): In this work, obfuscation is used to extend the session's lifetime

4 Enhancing the Security of Cloud Computing Using Obfuscation and Diversification

4.1 Motivation Behind Our Idea

As discussed in Sect. 2.1 about the importance of application security in cloud computing environment and the big losses that might happen as the consequence of insecure applications, we were motivated to propose an approach to improve the cloud's security through securing the applications. Considering the fact that obfuscation and diversification techniques have shown success in impeding the malware in various domains to lessen the risk of harmful damage, we were motivated to use this techniques in our approach. To this aim, first we investigated "how these two techniques are used in cloud computing with the goal of boosting security" (Hosseinzadeh et al. 2015). We systematically reviewed all the studies that were trying to answer this research question. By answering this question, we were aiming at identifying the research gaps which lead us in our future research. After collecting and analyzing the data, we concluded that: there is a growing interest in this field of study, as the number of publications was increasing year by year. Obfuscation and diversification techniques have been used in the literature in different ways, that we presented a classification of this studies based on the way they use these techniques. The classification is presented in Sect. 3.1. Furthermore, as the result of this survey, we realized that the majority of the studied works have proposed approaches using obfuscation techniques, and few were focusing on diversification techniques. This implies that there is a room for more research on the use of diversification as a beneficial technique to bring security to cloud computing.

The previous survey shed more light on the areas that are still potential targets for further research, which motivated us to propose an efficient approach with the help of diversification and obfuscation techniques to secure the cloud's applications. We discuss the details of the proposed approach in Sects. 4.3.

4.2 Threat Model

To make using and deploying SaaS applications easy, these applications are usually available in web environment. A significant proportion of the code of these applications is usually run on the client side, which makes them vulnerable to client-side attacks. Also, the client-side interfaces are often a natural weak point that an adversary can utilize to launch an attack. In what follows, we will concentrate on this threat.

In a man-in-the-browser attack (MitB), the adversary has successfully compromised the client's endpoint application, usually the browser, by getting malware into user's system. The malware can then modify how the browser represents certain web sites and how the user can interact with them. Because the malware is operating inside user's browser, it is able to perform actions using user's authentication credentials by exploiting active log-in sessions (Gühring 2006; Laperdrix et al. 2015).

To be more specific, the malicious program infects the computer's software. The malware – often implemented as a browser extension – then waits for the user to submit

some interesting data. As the data is input in the application, the malware intercepts this delivery and extracts all the data using the interfaces provided by the browser (usually by accessing the DOM interface using JavaScript) and stores the values. The malware then modifies the values using browser's interface. The malware then tells the browser to continue submitting the data to the server (or just store it locally in the web application) and the browser goes on without knowing the data has been tampered with. The modified values are now stored by the server (or locally), but neither the user nor the server knows that they are not the original values.

In the case the server generates a receipt of the performed transaction or otherwise shows the previously sent values to the user, the malware again transforms them to the original ones. The user thinks everything is fine, because it appears that the original transaction was received and stored intact. In reality, however, the stored values have been fabricated by the malicious adversary.

It is important to note that attacks of this kind have been seen in the wild (Binsalleeh et al. 2010) and there is no completely satisfactory solution to prevent them. Therefore, mitigating these attacks has become an important goal in the field of information security.

4.3 Our Proposed Approach

For this work, we decided to evaluate integrating source code level obfuscation into an existing web application written in JavaScript. Our solution is a proactive and transparent method that protects applications from data manipulation. Although it does not guarantee to completely prevent all tampering, it significantly mitigates the attack scenario we described.

An important key observation in our solution is the fact that a malicious program in the user's browser needs knowledge about the web application's internal structure in order to modify the data provided by the user. We therefore, change the application that is being executed on the user's web browser in a way that will make it very difficult for a harmful program to compromise it.

After we have applied unique obfuscation to the program, the code is unique on each user's computer. Generic and automatic large-scale malware attacks become infeasible, since the adversary needs to know what to change in the target application's code.

Given enough time, however, the attacker may be able to break the obfuscation. Taking this possibility into account, we could make attacking the web application even harder by continuously re-obfuscating it during its execution. As the internal structure of the web application is dynamically changed like this, a malicious program has only a short time to analyze it in order to modify the data.

In web environment, certain obfuscation methods can also be used to obfuscate the HTML code on the web page that is the target of protection. This makes it even harder for a piece of malware to attach itself to the web application (for example, by using known attribute names of HTML elements). In is also worth noting that in our scheme, we scramble HTML and JavaScript code but not to the actual data that is transmitted over the network. The usual cryptographic protocols like Transport Layer Security (TLS) (Dierks 2008), are still applied to this data on most web pages handling private data.

It is worth noting that when the obfuscation has been performed, the user of a web application will not notice any changes in the functionality of the application. Obfuscation is transparent to the user. We also want our solution to be transparent for the web application developer. The obfuscation is performed automatically after the code is written so the developer does not have to worry about it.

Data modification attacks are often highly dependent on the known structure of a web application. For example, the adversary might try to edit some function in the JavaScript code based on its known name. Our approach should therefore effectively mitigate these kinds of attacks by obscuring the structure of executable code.

4.4 Choice of Application

For the choice of application, we had the following criteria that the selected application had to fulfill:

1. Availability of production-ready obfuscation tooling and libraries. It can be argued, that source code level obfuscation tooling is still in its infancy and, at least in our experience gathered from this exercise, such tools are simply non-existent for many languages and environments. However, for some languages and environments – like JavaScript run in the user's browser – several obfuscation tools and libraries exist today.
2. The application had to be implemented using technologies that are common in today's web development environment. Using commonplace technologies was especially important because we wanted the experiences to be applicable to real-world web application deployment scenarios. In short, we wanted our choice of application be representative of a generic web application.

In the end, we decided to obfuscate Laverna (lav 2016), a simple note taking application that relies entirely for client side scripting for its functionality. Since Laverna contains essentially no server-side components, the main security risk it faces comes from man-in-the-browser attacks.

4.5 Implementation

Ideally, we would like the obfuscation to be seamlessly integrated into project's development work-flow. It is common for modern web-applications already contain a complex build process: resource compression, source code transpiling and request count optimization are just few of the steps that a typical application might employ. Orchestrating all these interdependent operations is a challenging task that has given a rise for a cornucopia of different build automation tools targeting the web platforms.

Laverna is not an exception on this front. The project makes heavy use of (gul 2016a), Browserify (bro 2016), and npm (npm 2016) to automate its build process and manage the complex web of dependencies required for building the application. As a part of the standard build process, Gulp transpiles stylesheets written in Less (les 2016)

into css, compresses the html, produces caching manifest and runs various code quality checkers on the project.

We wanted the obfuscation to be as transparent to the software developer as possible. To this end, we decided to integrate the source code obfuscation as an additional step in Gulp's project build specification. Using common tools for the deployment process served our overall goal: evaluating the real-world challenges related to deploying obfuscation.

The concrete obfuscation implementation is composed of three third-party components: gulp-js-obfuscator (gul 2016b), js-obfuscator (jso 2016), and the service provided by javascriptobfuscator.com (jav 2016). The last of which, provides the actual source code transformations in a *software-as-a-service* like manner. js-obfuscate implements a programmatic api around the the service and gulp-js-obfuscate provides integration with Gulp's build pipeline architecture.

The results of the diversification experiment we performed on Laverna applications with our tool indicate that the program would indeed be much more difficult to understand and tamper with after diversification has been applied. For example, even with the relatively simple obfuscation transformations our tool used, the median of Halstead difficulty (the difficulty of understanding a given program) was 54.1% larger for the functions of the diversified version of the program than for the original code. Naturally, even better results would be achieved with a framework that would use a larger set of even more resilient obfuscation transformations.

4.6 Limitations of the Approach

Tooling for analyzing errors in program code is obviously important from a software development standpoint and when it comes to web development, most popular browsers come with built-in debugging capabilities. Setting break-points, single-stepping through the program code, and inspecting objects are common requirements. Unfortunately, application of source code level obfuscation makes utilizing available tooling challenging, to say the least. The problem arises because the developer interacts with the original, unobfuscated source code, but the browser only has access to the obfuscated version of the code. This is not a problem that only affects obfuscation related tooling, source-to-source transpilers have long faced similar problems. However, it could be argued that the problem is magnified for by the very nature of obfuscation, desire to make programs harder to understand.

Source maps is a technique created to solve the aforementioned problem of debugging (sou 2016). Source maps provide the browser with auxiliary debugging information about the obfuscated scripts, allowing it to map the executed statements to statements in the original source. Unfortunately, our current setup did not provide support for source maps. This problem can be somewhat remedied by applying obfuscation only to release builds of the software. While this approach works, with the added benefit of making the build process faster, it does not allow analyzing problems that might arise due to the application of obfuscation. It also limits software developer's ability to analyze possible error reports from end users.

When obfuscating any application, preserving good performance is also an important goal and a challenge. Because of the requirement for transparency, large performance losses clearly noticed by the user are not acceptable. As a response to increasing use of JavaScript frameworks and ongoing competition between web browser manufacturers, performances of the JavaScript engines have gone up in recent years. Acceptable performance and good transparency to the user are usually feasible goals even when using several obfuscation techniques in combination and the obfuscation is dynamically changed.

Employing obfuscation also often increases bandwidth consumption as the executable code grows longer. Dynamically updating the code during execution – a feature not implemented in our current proof-of-concept implementation – would also significantly increase the network traffic. All obfuscation techniques do not increase the size of code that much, though. For example, simply renaming functions does not really affect the bandwidth consumption.

The SaaS-based obfuscation backend provided by javascriptobfuscator.com supports a number of obfuscating transformations. The basic settings employ standard techniques such as variable renaming and string encoding, but more complicated transformation options are also available. Still, we felt that the service-oriented solution limited the amount of control over how the code was to be modified. Figure 4 gives an idea of what the end result of the obfuscation looks like.

```
744974656D"];define([_0xa6ab[0],_0xa6ab[1],_0xa6ab[2]],function(_0x27cax1,_0x27cax2,_0x
27cax3)_0xa6ab[3];var _0x27cax4=dbs:,getDb:function(_0x27cax5)var
_0x27cax6=_0x27cax5[_0xa6ab[4]]+_0xa6ab[5]+_0x27cax5[_0xa6ab[6]];this[_0xa6ab[7]][_0x
27cax6]=this[_0xa6ab[7]][_0x27cax6]||_0x27cax3[_0xa6ab[9]](name:_0x27cax5[_0xa6ab[4]]||
_0xa6ab[8],storeName:_0x27cax5[_0xa6ab[6]]);return
this[_0xa6ab[7]][_0x27cax6],find:function(_0x27cax7)var
_0x27cax8=_0x27cax2[_0xa6ab[10]]();this[_0xa6ab[17]](_0x27cax7[_0xa6ab[16]])[_0xa6ab[1
5]](_0x27cax7[_0xa6ab[11]],function(_0x27cax9,_0x27cax7)if(_0x27cax9)return
_0x27cax8[_0xa6ab[12]](_0x27cax9);if(!_0x27cax7)_0x27cax8[_0xa6ab[12]](_0xa6ab[13]);ret
urn _0x27cax8[_0xa6ab[14]](_0x27cax7));return
_0x27cax8[_0xa6ab[18]],findAll:function(_0x27cax7)var
_0x27cax8=_0x27cax2[_0xa6ab[10]](),_0x27caxa=this;this[_0xa6ab[17]](_0x27cax7[_0xa6ab[
16]])[_0xa6ab[23]](function(_0x27cax9,_0x27caxb)if(!_0x27caxb||!_0x27caxb[_0xa6ab[19]])re
turn _0x27cax8[_0xa6ab[14]]([]);
```

Fig. 4. Excerpt from obfuscated piece of JavaScript code.

5 Conclusion

Cloud computing is becoming an essential part of today's Information Technology. Almost all enterprises and businesses, in all sizes, have deployed (or are planning to deploy) cloud solutions for delivering their services to customers. Cloud adoption is accelerating because of the advantages that cloud computing has brought along, such as higher flexibility and capability of the infrastructures, lower costs of operation and

maintenance, wider accessibility, and improved mobility and collaboration (Mather et al. 2009).

Despite of all these benefits, there are still barriers in turning into cloud. Among all, security of the data is the primary concern that holds back the projects from moving to the cloud. The cloud's security threats can be classified in different ways. Cloud Security Alliance (CSA) presented a list of top threats targeting the cloud computing environment (CSA 2016; Top Threats Working Group 2013).

In Sect. 2, we went through the security concerns of the cloud and also security aspects that need to be taken into account in cloud computing environment. We discussed that there are three main delivery models for delivering the cloud services (IaaS, PaaS, and SaaS) that each require different levels of security (Rhoton et al. 2013).

In Sect. 3, first we presented the terms and techniques used in our proposed security approach. Obfuscation and diversification are techniques that have been used to secure the software, mainly with the aim of impeding malware. These techniques have been utilized in various domains as well as in cloud computing. In a previous study we conducted a thorough survey to investigate in what way these two techniques have been previously used to enhance the security of cloud computing and protect the privacy of its users (Hosseinzadeh et al. 2015). As the result of this study, we managed to identify research gaps that motivated us to demonstrate an approach, which fills the gaps to some extent and improves the security in cloud efficiently.

In Sect. 4 we demonstrated an obfuscation (partly including diversification) approach for mainly securing the SaaS model in cloud computing. In this approach we obfuscated the client-side JavaScript components of an application, we did this to demonstrate the feasibility of applying obfuscation in the real-world. We built our solution using existing tools and services to evaluate the experience of integrating obfuscation into an existing application. Implementing the obfuscation only required a relatively small amount of work, mostly because of the use of ready-made libraries. However, the amount of work required is likely to be highly dependant on the complexity of one's target application and the thoroughness of applied obfuscation.

References

Browserify (2016). http://browserify.org. Accessed 08 Apr 2016

Cloud Security Alliance (CSA) (2016). https://cloudsecurityalliance.org/. Accessed 08 Apr 2016

Free JavaScript obfuscator Protect JavaScript code from stealing and shrink size (2016). https://javascriptobfuscator.com. Accessed 08 Apr 2016

Getting started–Less.js (2016). http://lesscss.org. Accessed 08 Apr 2016

Gulp-js-obfuscator (2016a). https://www.npmjs.com/package/gulp-js-obfuscator. Accessed 08 Apr 2016

Gulp.js The streaming build system (2016b). http://gulpjs.com. Accessed 08 Apr 2016

js-obfuscator (2016). https://www.npmjs.com/package/js-obfuscator. Accessed 08 Apr 2016

Laverna Keep your notes private (2016). https://laverna.cc. Accessed 08 Apr 2016

NMP (2016). https://www.npmjs.com. Accessed 08 Apr 2016

Source Map Revision 3 Proposal (2016). https://docs.google.com/document/d/1U1RGAehQwRypUTovF1KRlpiOFze0b-2gc6fAH0KY0k. Accessed 08 Apr 2016

The International Information Systems Security Certification Consortium (ISC)^2 (2016). https://www.isc2.org/. Accessed 08 Apr 2016

Agir, B., Papaioannou, T., Narendula, R., Aberer, K., Hubaux, J.-P.: User-side adaptive protection of location privacy in participatory sensing. GeoInformatica **18**(1), 165–191 (2014)

Arockiam, L., Monikandan, S.: Efficient cloud storage confidentiality to ensure data security. In: 2014 International Conference on Computer Communication and Informatics (ICCCI), pp. 1–5 (2014)

Baudry, B., Monperrus, M.: The multiple facets of software diversity: recent developments in year 2000 and beyond. ACM Comput. Surv, **48**(1), 16:1–16:26 (2015)

Bertholon, B., Varrette, S., Bouvry, P.: JShadObf: a JavaScript obfuscator based on multi-objective optimization algorithms. In: Lopez, J., Huang, X., Sandhu, R. (eds.) NSS 2013. LNCS, vol. 7873, pp. 336–349. Springer, Heidelberg (2013). doi:10.1007/978-3-642-38631-2_25

Bertholon, B., Varrette, S., Bouvry, P.: Comparison of multi-objective optimization algorithms for the Jshadobf JavaScript obfuscator. In: 2014 IEEE International, Parallel Distributed Processing Symposium Workshops (IPDPSW), pp. 489–496 (2014)

Bertholon, B., Varrette, S., Martinez, S.: Shadobf: A c-source obfuscator based on multi-objective optimization algorithms. In: 2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), pp. 435–444 (2013b)

Binsalleeh, H., Ormerod, T., Boukhtouta, A., Sinha, P., Youssef, A., Debbabi, M., Wang, L.: On the analysis of the zeus botnet crimeware toolkit. In: Proceedings of the 8th Annual International Conference on Privacy, Security and Trust (PST), pp. 31–38. IEEE (2010)

Celesti, A., Fazio, M., Villari, M., Puliafito, A.: Adding long-term availability, obfuscation, and encryption to multi-cloud storage systems. J. Netw. Comput. Appl. (2014)

Chang, V.: Towards a big data system disaster recovery in a private cloud. Ad Hoc Netw. **35**, 65–82 (2015). Special Issue on Big Data Inspired Data Sensing, Processing and Networking Technologies

Chang, V., Kuo, Y.-H., Ramachandran, M.: Cloud computing adoption framework: a security framework for business clouds. Future Gener. Comput. Syst. **57**, 24–41 (2016)

Chang, V., Ramachandran, M.: Towards achieving data securCloud computing adoption framework: a security framework for business cloudsity with the cloud computing adoption framework. IEEE Trans. Serv. Comput. **9**(1), 138–151 (2016)

Chen, T.M., Abu-Nimeh, S.: Lessons from stuxnet. Computer **44**(4), 91–93 (2011)

Cohen, F.B.: Operating system protection through program evolution. Comput. Secur. **12**(6), 565–584 (1993)

Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand (1997)

Collberg, C., Thomborson, C., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1998, pp. 184–196. ACM, New York (1998)

Dierks, T.: The Transport Layer Security (TLS) protocol version 1.2 (2008)

Drape, S., Majumdar, A.: Design and evaluation of slicing obfuscation. Technical report, Department of Computer Science, The University of Auckland, New Zealand (2007)

Furukawa, R., Takenouchi, T., Mori, T.: Behavioral tendency obfuscation framework for personalization services. In: Decker, H., Lhotská, L., Link, S., Basl, J., Tjoa, A.M. (eds.) DEXA 2013. LNCS, vol. 8056, pp. 289–303. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40173-2_24

Gao-xiang, G., Zheng, Y., Xiao, F.: The homomorphic encryption scheme of security obfuscation. In: Tan, T., Ruan, Q., Chen, X., Ma, H., Wang, L. (eds.) IGTA 2013. CCIS, vol. 363, pp. 127–135. Springer, Heidelberg (2013). doi:10.1007/978-3-642-37149-3_16

Govinda, K., Sathiyamoorthy, E.: Agent based security for cloud computing using obfuscation. Procedia Eng. **38**, 125–129 (2012)

Gühring, P.: Concepts against Man-in-the-Browser Attacks (2006). www.cacert.at/svn/sourcerer/CAcert/SecureClient.pdf

Guo, M., Bhattacharya, P.: Diverse virtual replicas for improving intrusion tolerance in cloud. In: Proceedings of the 9th Annual Cyber and Information Security Research Conference, CISR 2014, pp. 41–44. ACM, New York (2014)

Hataba, M., El-Mahdy, A.: Cloud protection by obfuscation: techniques and metrics. In: 2012 Seventh International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), pp. 369–372 (2012)

Hosseinzadeh, S., Hyrynsalmi, S., Conti, M., Leppänen, V.: Security and privacy in cloud computing via obfuscation and diversification: a survey. In: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), pp. 529–535 (2015)

Kansal, K., Mohanty, M., Atrey, Pradeep, K.: Scaling and cropping of wavelet-based compressed images in hidden domain. In: He, X., Luo, S., Tao, D., Xu, C., Yang, J., Hasan, M.A. (eds.) MMM 2015. LNCS, vol. 8935, pp. 430–441. Springer, Heidelberg (2015). doi:10.1007/978-3-319-14445-0_37

Karuppanan, K., AparnaMeenaa, K., Radhika, K., Suchitra, R.: Privacy adaptation for secured associations in a social cloud. In: 2012 International Conference on Advances in Computing and Communications (ICACC), pp. 194–198 (2012)

Kuzu, M., Islam, M. S., Kantarcioglu, M.: Efficient privacy-aware search over encrypted databases. In: Proceedings of the 4th ACM Conference on Data and Application Security and Privacy, CODASPY 2014, pp. 249–256. ACM, New York (2014)

Lamanna, D.D., Lodi, G., Baldoni, R.: How not to be seen in the cloud: a progressive privacy solution for desktop-as-a-service. In: Meersman, R., Panetto, H., Dillon, T., Rinderle-Ma, S., Dadam, P., Zhou, X., Pearson, S., Ferscha, A., Bergamaschi, S., Cruz, I.F. (eds.) OTM 2012. LNCS, vol. 7566, pp. 492–510. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33615-7_4

Laperdrix, P., Rudametkin, W., Baudry, B.: Mitigating browser fingerprint tracking: multi-level reconfiguration and diversification. In: 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), pp. 98–108 (2015)

Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: 2014 IEEE Symposium on Security and Privacy (SP), pp. 276–291 (2014)

Laurén, S., Mäki, P., Rauti, S., Hosseinzadeh, S., Hyrynsalmi, S., Leppänen, V.: Symbol diversification of Linux binaries. In: Proceedings of World Congress on Internet Security (WorldCIS-2014) (2014)

Li, L., Li, Q., Shi, Y., Zhang, K.: A new privacy-preserving scheme DOSPA for SaaS. In: Gong, Z., Luo, X., Chen, J., Lei, J., Wang, F. (eds.) Web Information Systems and Mining. LNCS, vol. 6987, pp. 328–335. Springer, Berlin Heidelberg (2011)

Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, pp. 290–299. ACM, New York (2003)

Liu, X., Yuan, D., Zhang, G., Li, W., Cao, D., He, Q., Chen, J., Yang, Y.: Cloud workow system quality of service. In: The Design of Cloud Workow Systems, Springer Briefs in Computer Science, pp. 27–50. Springer, New York (2012)

Mather, T., Kumaraswamy, S., Latif, S.: Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance Theory in Practice. O'Reilly Media Inc., Sebastopol (2009)

Mell, P., Grance, T.: The NIST definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology (2011)

Mowbray, M., Pearson, S.: A client-based privacy manager for cloud computing. In: Proceedings of the Fourth International ICST Conference on Communication System software and middleware, COMSWARE 2009, pp. 5:1–5:8. ACM, New York (2009)

Mowbray, M., Pearson, S., Shen, Y.: Enhancing privacy in cloud computing via policy-based obfuscation. J. Supercomput. **61**(2), 267–291 (2012)

Nagra, J., Collberg, C.: Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Pearson Education, Upper Saddle River (2009)

Omar, R., El-Mahdy, A., Rohou, E.: Arbitrary control-ow embedding into multiple threads for obfuscation: a preliminary complexity and performance analysis. In: Proceedings of the 2nd International Workshop on Security in Cloud Computing, SCC 2014, pp. 51–58. ACM, New York (2014)

Padilha, R., Pedone, F.: Confidentiality in the cloud. Secur. Privacy IEEE **13**(1), 57–60 (2015)

Palanques, M., DiPietro, R., del Ojo, C., Malet, M., Marino, M., Felguera, T.: Secure cloud browser: model and architecture to support secure web navigation. In: 2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS), pp. 402–403 (2012)

Patibandla, R.,S.,M.,Lakshmi, Kurra, S.S., Mundukur, N.B.: A study on scalability of services and privacy issues in cloud computing. In: Ramanujam, R., Ramaswamy, S. (eds.) ICDCIT 2012. LNCS, vol. 7154, pp. 212–230. Springer, Heidelberg (2012). doi:10.1007/978-3-642-28073-3_19

Pearson, S., Shen, Y., Mowbray, M.: A privacy manager for cloud computing. In: Jaatun, M.G., Zhao, G., Rong, C. (eds.) CloudCom 2009. LNCS, vol. 5931, pp. 90–106. Springer, Heidelberg (2009). doi:10.1007/978-3-642-10665-1_9

Popov, I.V., Debray, S.K., Andrews, G.R.: Binary obfuscation using signals. In: USENIX Security (2007)

Prasadreddy, P., Rao, T., Venkat, S.: A threat free architecture for privacy assurance in cloud computing. In: 2011 IEEE World Congress on Services (SERVICES), pp. 564–568 (2011)

Qin, Y., Shen, S., Kong, J., Dai, H.: Cloud-oriented SAT solver based on obfuscating CNF formula. In: Han, W., Huang, Z., Hu, C., Zhang, H., Guo, L. (eds.) APWeb 2014. LNCS, vol. 8710, pp. 188–199. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11119-3_18

Rauti, S., Laurén, S., Hosseinzadeh, S., Mäkelä, J.-M., Hyrynsalmi, S., Leppänen, V.: Diversification of system calls in Linux binaries. In: Proceedings of the 6th International Conference on Trustworthy Systems (In Trust 2014) (2014)

Reiss, C., Wilkes, J., Hellerstein, J.: Obfuscatory obscanturism: making workload traces of commercially-sensitive systems safe to release. In: 2012 IEEE Network Operations and Management Symposium (NOMS), pp. 1279–1286 (2012)

Rhoton, J., de Clercq, J., Graves, D.: Cloud Computing Protected: Security Assessment Handbook. Recursive Limited, London (2013)

Ryan, P., Falvey, S.: Trust in the clouds. Comput. Law Secur. Rev. **28**(5), 513–521 (2012)

Skoudis, E.: Malware: Fighting Malicious Code. Prentice Hall Professional, Upper Saddle River (2004)

Skvortsov, P., Dürr, F., Rothermel, K.: Map-aware position sharing for location privacy in non-trusted systems. In: Kay, J., Lukowicz, P., Tokuda, H., Olivier, P., Krüger, A. (eds.) Pervasive 2012. LNCS, vol. 7319, pp. 388–405. Springer, Heidelberg (2012). doi:10.1007/978-3-642-31205-2_24

Subashini, S., Kavitha, V.: A survey on security issues in service delivery models of cloud computing. J. Netw. Comput. Appl. **34**(1), 1–11 (2011)

Tapiador, J., Hernandez-Castro, J., Peris-Lopez, P.: Online randomization strategies to obfuscate user behavioral patterns. J. Netw. Syst. Manag. **20**(4), 561–578 (2012)

Tian, Y., Song, B., Huh, E.-N.: Towards the development of personal cloud computing for mobile thin-clients. In: International Conference Information Science and Applications (ICISA), pp. 1–5 (2011)

Top Threats Working Group: The notorious nine: cloud computing top threats in 2013. Cloud Security Alliance (2013)

Tunc, C., Fargo, F., Al-Nashif, Y., Hariri, S., Hughes, J.: Autonomic resilient cloud management (ARCM) design and evaluation. In: 2014 International Conference on Cloud and Autonomic Computing (ICCAC), pp. 44–49 (2014)

Varadharajan, V., Tupakula, U.: Security as a service model for cloud environment. IEEE Trans. Netw. Serv. Manag. **11**(1), 60–75 (2014)

Villari, M., Celesti, A., Tusa, F., Puliafito, A.: Data reliability in multi-provider cloud storage service with RRNS. In: Canal, C., Villari, M. (eds.) Advances in Service-Oriented and Cloud Computing. Communications in Computer and Information Science, vol. 393, pp. 83–93. Springer, Heidelberg (2013)

Vleju, M.B.: A client-centric ASM-based approach to identity management in cloud computing. In: Castano, S., Vassiliadis, P., Lakshmanan, Laks, V., Lee, M.L. (eds.) ER 2012. LNCS, vol. 7518, pp. 34–43. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33999-8_5

Yang, P., Gui, X., Tian, F., Yao, J., Lin, J.: A privacy-preserving data obfuscation scheme used in data statistics and data mining. In: High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC-EUC), pp. 881–887 (2013)

Yang, Q., Cheng, C., Che, X.: A cost-aware method of privacy protection for multiple cloud service requests. In: 2014 IEEE 17th International Conference on Computational Science and Engineering (CSE), pp. 583–590 (2014)

Yau, S.S., An, H.G.: Protection of users' data confidentiality in cloud computing. In: Proceedings of the Second Asia-Pacific Symposium on Internetware, Internetware 2010, pp. 11:1–11:6. ACM, New York (2010)

Zhang, G., Liu, X., Yang, Y.: Time-series pattern based effective noise generation for privacy protection on cloud. IEEE Trans. Comput. **64**(5), 1456–1469 (2015)

Zhang, G., Yang, Y., Chen, J.: A historical probability based noise generation strategy for privacy protection in cloud computing. J. Comput. Syst. Sci. **78**(5), 1374–1381 (2012a). {JCSS} Special Issue: Cloud Computing 2011

Zhang, G., Yang, Y., Chen, J.: A privacy-leakage-tolerance based noise enhancing strategy for privacy protection in cloud computing. In: 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 1–8 (2013)

Zhang, G., Yang, Y., Liu, X., Chen, J.: A time-series pattern based noise generation strategy for privacy protection in cloud computing. In: 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 458–465 (2012b)

Zhang, G., Yang, Y., Yuan, D., Chen, J.: A trust-based noise injection strategy for privacy protection in cloud. Softw.: Pract. Exp., **42**(4), 431–445 (2012c)

Zhang, G., Zhang, X., Yang, Y., Liu, C., Chen, J.: An association probability based noise generation strategy for privacy protection in cloud computing. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) ICSOC 2012. LNCS, vol. 7636, pp. 639–647. Springer, Heidelberg (2012b). doi:10.1007/978-3-642-34321-6_50

Publication III

Security in Container-based Virtualization through vTPM

Shohreh Hosseinzadeh, Samuel Laurén, Ville Leppänen. In: 2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing (UCC 2016), Pages 214-219. IEEE/ACM 2016.

2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing

Security in Container-based Virtualization through vTPM

Shohreh Hosseinzadeh, Samuel Laurén, and Ville Leppänen
Department of Information Technology
University of Turku, Finland
{shohos, samuel.lauren, ville.leppanen}@utu.fi

ABSTRACT

Cloud computing is a wide-spread technology that enables the enterprises to provide services to their customers with a lower cost, higher performance, better availability and scalability. However, privacy and security in cloud computing has always been a major challenge to service providers and a concern to its users. Trusted computing has led its way in securing the cloud computing and virtualized environment, during the past decades.

In this paper, first we study virtualized trusted platform modules and integration of vTPM in hypervisor-based virtualization. Then we propose two architectural solutions for integrating the vTPM in container-based virtualization model.

CCS Concepts

•**Security and privacy** → **Systems security;** *Trusted computing;* Virtualization and security;

Keywords

cloud computing, security, trusted computing, trusted platform module, TPM, vTPM

1. INTRODUCTION

With the advancement in cloud computing technologies, more and more enterprises and service providers are shifting towards this technology and delivering their services over clouds. This is while security has always been a challenge in cloud computing technology. Due to the nature of cloud computing, the data is stored outside the perimeters of the organizations, which brings along concerns about the data security and the privacy. There is a large body of research on securing the cloud computing through various techniques [11], [21]. Cloud service providers are always attempting to protect the cloud from insider and outsider attacks through security measures such as controlling the access to software and hardware facilities. Nonetheless, due to the fact that there is still some level of access from the administrative level to user's Virtual Machine (VM), there is still a need for protecting the integrity and confidentiality of computation in the cloud [20].

Trusted Computing technology, developed by Trusted Computing Group (TCG) [3], is a technology to ensure that the computers behave in expected ways. This technology provides a hardware based security solution through Trusted Platform Module (TPM) [4]. Typically, it contains a physical chip embedded on the motherboard of a platform and is controlled via a software.

Trusted computing is a mechanism to establish a secure environment and provide privacy and trust. The trusted computing mechanism can as well be extended to cloud computing environment by integrating the trusted platform into cloud architecture. The trusted platform is integrated into the cloud through TPM and provides the cloud with secure functionalities such as secure authentication, access control, communication security, and data protection [22]. For instance, integrating the trusted computing to IaaS is a way to certify the confidential execution of the VMs running on the platform [20].

In this paper, we first discuss two different virtualization technologies and present a comparison between these two models in terms of security and efficiency. Then we study how trusted computing technology is extended to cloud computing and virtual environments with the aim of improving the security. We propose solutions for integrating TPM to container-based virtualization model for improving security and also supporting secure live migration of the virtual machines. To the best of our knowledge, we are not aware of any previous virtual TPM constructs for container-based virtualization.

The remainder of this paper is structured as follows. Section **??** presents an introductory background on the terms and concepts related to this research work. Section 2 overviews the related works in this domain. In Section 3 we present our virtual TPM solutions for the container setting. Conclusions and future work proposals come in Section 4. **a) Trusted Platform Module (TPM)** is a micro-controller used in computing devices (e.g., PCs, servers, mobile phones, appliances) to provide hardware root of trust, for instance for identification of the user and device, data protection, network access [2]. TPM mainly presents cryptography functionalities, such as encryption, decryption, and signing cryptographic keys, and also stores the cryptographic keys. The artifacts (i.e., passwords, encryption keys, certificates) used for authentication of the device are stored on the TPM chip.

WISARC '16 Shanghai, China

DOI: 10.475/123_4

TPM is as well used for ensuring the trustworthiness of the platform. To this end, it stores platform measurements that ensure the safety of the computation. These measurements contain authentication (to assure that the platform is what it asserts to be), and attestation of the platform (to assure that the platform has not been corrupted/breached and is trustworthy).

b) Virtual Trusted Platform Module (vTPM): In the recent years hardware platform virtualization has been an advantageous approach in cutting the operation expenses by sharing the platform amongst several software workloads (e.g., web hosting centers). However, sharing the same platform brings security issues along which require regulations for separation of the workloads, the resources, and also ensuring software integrity. Virtual Machine Monitors (=hypervisors) are good solutions for the isolating of these workloads, and hardware based root of trust through TPM guarantees the integrity of software and mitigating the software attacks. Therefore, the combination of these two technologies is a well-fitting security solution in this scenario [7]. In order to make the TPM available to multiple operating systems concurrently, the idea of virtualized platform is used. TPM is emulated to each of the guest operating systems, in a way that they can interact with TPM and access its functionalities, as with a TPM on a physical system [5]. Each guest virtual machine accesses the unique emulated TPM, as it appears that there is a separate TPM for each platform and from the virtual machines' viewpoints they have their own TPM. Moreover, vTPM should provide all the functionalities that the physical TPM presents to the system. For instance, modern hypervisors enable the operating systems to migrate between the physical hosts. Thus, it is expected that the virtual TPMs support this feature as well [7].

c) Hypervisor-based virtualization vs container-based virtualization: During the past decades various virtualization mechanisms have been developed including hypervisor-based virtualization and container-based virtualization. In *hypervisor-based virtualization* a hypervisor (Virtual Machine Monitor (VMM)) is running on top of the server hardware enabling multiple virtual machines to share virtualized hardware resources. This means that different operating systems (e.g., Windows, Linux) can run on a physical platform. While, in *operating-system-level virtualization (also referred to as container-based virtualization)* rather than hardware, the virtualization happens at the operating system level. In other words, containers share the same operating system kernel. Thus, OS virtualization allows running multitude execution environment instances on a single kernel [1]. Figure 1 depicts the architecture of these two virtualization models.

Each of these technologies have their own strengths and weaknesses. OS-level virtualization has less performance overhead. As containers are more lightweight than the virtual machines and therefore, it is faster to boot a container comparing to a guest OS. On the other hand, OS-level virtualization has less flexibility in a way that it only hosts the same OS as the the host is using [19].

d) Virtual machine migration: Migration of virtual machines is the act of moving a virtual machine from one physical host to another. This ability is helpful in several different ways. For instance, a) for *load balancing*, i.e., to move the VM to a less loaded host, b) at the time of *maintenance and upgrades* when a server needs to be shut down

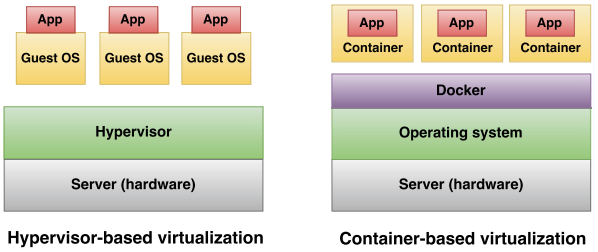


Figure 1: Hypervisor-based virtualization vs container-based

the VM could be transferred to another server, and c) in *disaster recovery*, when a host has failed the VM could be restarted from another host. Virtual machine migration can be done in offline or live modes. Offline migration is quite straightforward, i.e., the VM is turned off and then the migration process occurs from the source to the destination host. Provided that, live migration lets the VMs to migrate from a physical host to another while the VM is in use. I.e., no significant interruption happens to the availability of the VM and applications for thier users [15]. In spite of the advantages that the live migration presents to virtualization technology, its security still needs consideration. E.g., Oberheide [17] has discussed three different classes of security threats targeting the process of VM live migration. Concerning container-based virtualization, we note that there is very little literature on migration in that setting presently. On the other hand, hypervisor-based vTPM solutions focus on supporting migration, and we will also consider that in our solutions.

2. RELATED WORK

Berger et al. [7] have presented software implementation of virtualized TPM to make TPMs available for multiple virtual machines running on top of a hypervisor. Their design is shown in Figure 2. The cryptographic functions and secure storage services supported by TPM will be available to the applications and operating systems running on top of virtual machines. The TPM specification is implemented in software and the vTPM is placed into a dedicated VM. The vTPM provides support for migration of vTPM instances, suspension, and resume operations. However, a drawback of this approach (i.e., implementing the TPM in software) is that it cannot be capable of supporting the security needs and protection levels as well as the hardware does [25]. To overcome this limitation, Stumpf and Eckert [25] have proposed multi-context TPM approach that is completely hardware based and enables the VMs to benefit fully from TPM's functionalities. In this approach, virtual machine monitor uses a second privilege level to issue scheduling and management commands. Information about the state of a TPM is stored in TPM Control Structure (TPMCS), which ensures that the state of one VM's TPM does not corrupt the state of another VM's TPM. They also propose a migration protocol that supports secure migration of TPM contexts to a new TPM.

England and Loeser [9] consider two different studies [7, 12] that provide trusted computing in virtual environments, then they propose a para-virtualized TPM approach that

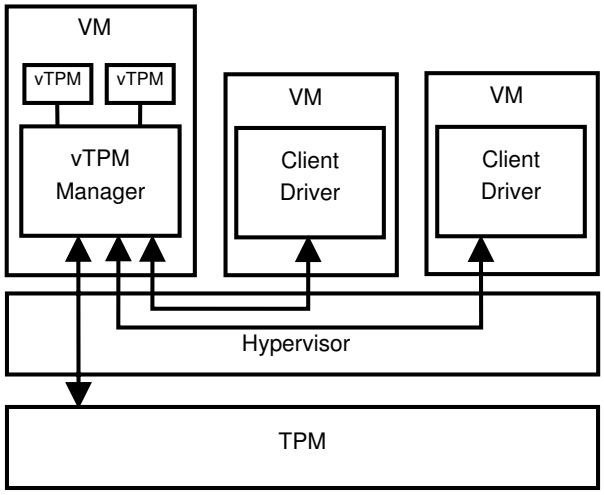


Figure 2: vTPM design proposed by Berger et al. [7]

shares the TPM (v1.2) among several operating systems. This approach is close to the both presented ideas in the two previous works. It multiplexes the TPM hardware and takes advantage of the TPM hardware key protection. Then it uses software components for safe device sharing.

Shi et al. [23] noted the difficulties associated with securing software based vTPMs and proposed an improved design where vTPM secrets are protected using symmetric encryption. Their solution implements vTPMs for QEMU based virtual machines by utilizing kernel and user space components. Wan et al. [26] analyzed existing vTPM solutions in order to better understand the security properties of different implementation approaches.

As discussed earlier, migration of the virtual machine is an advantageous feature supported in today's virtual environments. When migrating a VM, the related vTPM should be considered to move as well. Therefore, in the literature, there have been works done on secure migration of VM-vTPM. Hong et al. [14] have studied various VM-vTPM live migration protocols and analyzed them in terms of performance and security. Following that, they proposed a trusted live migration protocol based on pre-copy. This protocol includes three different phases: authentication and remote attestation of the source and destination platforms, and secure data transfer. The implementation is based on Xen [6]. Another vTPM-VM live migration protocol is proposed by Fan et al. [10] which consists of two different phases, integrity verification and data transfer. In the earlier phase, the source and destination platforms mutually authenticate themselves, negotiate a session key, and construct a secure channel. In the later phase, the memory content and all VM's external states are transferred. The security of this protocol is analyzed by studying how resistant it is against various types of attacks. For instance, the attacks may occur a) between the virtual machines, b) on communication between the host operating system and the virtual machine, and c) on data transmission channel. Danev et al. [8] consider the secure migration of vTPM-VM in private cloud environments. They study the security requirements of a secure migration, and propose a vTPM key structure and also a vTPM base VM migration protocol. By implementing the proposed scheme on Xen hypervisor, they claim that the protocol guarantees stronger security for private virtualized environments. Similarly, Liang et al. [16] propose a VM-vTPM migration protocol for private clouds, to guarantee the confidentiality, integrity, and authentication of the source and destination hosts. The protocol consists of three phases, first the source and destination platforms mutually authenticate themselves to each other. Second, the vTPM is migrated, and in third, the VM is migrated.

3. PROPOSED APPROACHES

Virtualization and trusted computing are both promising technologies. Combination of these two offers a well-favoured secure solutions for the many domains including cloud computing. Virtualization is a beneficial factor in implementing a trusted platform. Integrating a TPM can support the security in virtual environments. The main advantage that it presents is the process isolation, which prevents a software process conflicting with another. For instance, it offers secure key storage of the secrets on the platform, identification of the VMs to their guest operating systems, and remote attestations for the hypervisors [18].

As discussed earlier, there are two types of virtualization, hypervisor-based and container-based. Container-based virtualization (also known as OS-level virtualization) has the advantage of being faster and consuming less storage, while on the other hand has the limitation that the user cannot use several different types of operating systems. As the container-based model becomes more popular, security and management of the containers and also the hosting platform become important issues.

We studied the integration of the trusted computing technology into the virtual environments in Section 2 and realized that in all the studied research works, vTPM is proposed and implemented for hypervisor-based virtualization model, and none is considering solution for container-based virtualization. This motivated us to consider integrating vTPM in container-based virtual environments. To achieve this goal, we propose two different architectures that can integrate TPM hardware module in container-based virtualization model and virtualize it to be used by multiple containers.

1) vTPM in the operating system kernel: In this proposed architecture, the TPM module is placed below the container-layer into the OS kernel. In order to make the TPM available to several containers, it needs to be virtualized. In our design, a kernel module is used to provide arbitrary number of software based vTPMs. Like hardware based TPMs, the software TPM presents a character device type interface to the userspace. These vTPMs can be exposed to containers through the usual mechanisms. The vTPMs are linked with the physical TPM. Figure 3 illustrates this architecture.

In order to evaluate a vTPM architecture we refer to the four requirements proposed by Berger et al. [7]:

R1 vTPMs must offer the same interface to the software as hardware based TPM does, that is, the use of vTPM must be transparent to software.

R2 There must exist a strong association between the vTPM and the virtual machine.

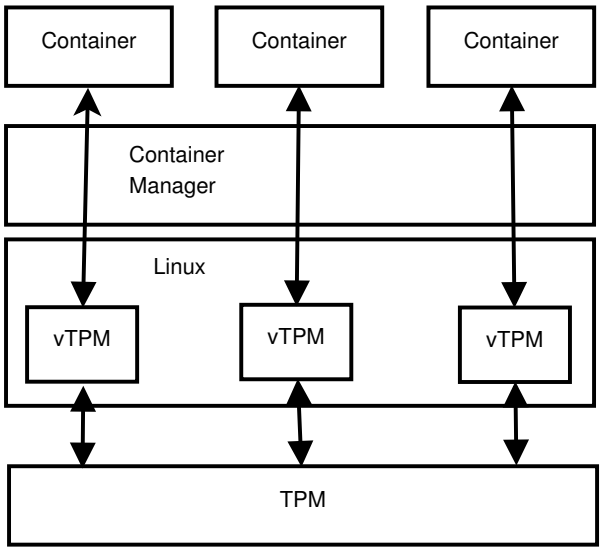


Figure 3: vTPM implemented in a kernel module

R3 There must exist a strong association between the vTPM and the trusted computing base.

R4 vTPM must be distinguishable from the physical TPMs. This is important because of the potentially different security guarantees associated with the different TPM types.

As per Requirement R1, the interface provided by our proposed vTPM is equivalent to the one provided by a physical TPM. From a software point of view, the character devices are identical.

However, as Berger et al. [7] have noted, implementing the low-level cryptographic operations is not necessarily the challenging part of TPM virtualization. The challenge is providing comparable security guarantees to hardware based trusted platform modules.

Assuming the container system providing the container isolation fulfills the requirements introduced by Reshetova et al. [19], we can consider the security properties provided by our in-kernel implementation from the viewpoint of a container. If containers cannot modify the host kernel (e.g. by loading new modules or by exploiting vulnerability in the kernel), they can reliably attest their own state by using hash extend feature of the vTPM. This however, assumes that the host operating system running below the containers is trusted. In the ideal case, we would like the container to be able to verify, not only its own state, but the state of the OS running the container. The trust should be rooted on hardware as R4 mandates.

This challenge of allowing the virtualized (or in our case, containerized) VM to verify the platform it is running on is also touched by the solution presented by Berger et al. [7]. Their solution for this is two-fold. In their implementation, the system exposes some of its platform configuration registers to the virtual machine, this allows the virtual machine to have a view of the state of the underlying platform. However, it is worth to note that this solution is problematic for migration since information sealed with the PCR values belonging to a different physical machine cannot be unsealed after migration.

Simply, having access to the supposed state of the underlying hardware is not enough. The vTPM itself has to be trusted. On a physical TPM, this is done by having a platform provider sign an endorsement key (EK) stating that the TPM is trustworthy. Berger et al. [7] extend this by having each TPM have its own endorsement key. They propose multiple different protocols for signing the endorsement keys of vTPMs with the help of the hardware based TPM. Each of the schemes has different security trade-offs associated with it.

However, the details of the scheme used to anchor the trust to a physical TPM are not the main focus of our proposal. We believe that multiple solutions could exists for these problems and many of the schemes presented in the literature targeting hardware hypervisors could also be applied in a container setting.

However, like the proposal presented in [7], many of the existing schemes for vTPM assume the hypervisor is responsible for managing all hardware access of the VMs and able to prevent unauthorized access to the software based vTPM. When borrowing ideas originally targeted to be used in full or even para-virtualized systems to containers, one has to consider the security implications that arise as the hypervisor is removed from the setting.

There have been multiple analyses of container security and much have been written about their security properties in contrast to a more traditional hypervisor-based approach [13, 19]. Many of the challenges come down to the multitude of interfaces exposed by container-based solutions and the general newness of the code maintaining the isolation of containers.

By putting the software based TPM inside the kernel, we can better protect it from unauthorized access by other processes and containers, than we could in a user-space based solution. However, this level of protection is arguably still less than the one provided by a full hypervisor.

In regards to the Requirement R2, a container manager would most likely be responsible for asking the kernel to create a new vTPM and assigning the device to a container. This could be done at the time the container is created.

2) vTPM placed in a separate container: Figure 4 illustrates this architecture. This solution is in some ways, even more similar to the design originally proposed by Berger et al. [7]. Where in their design, the virtual TPM manager is placed inside a separate Xen domain, here we have placed the software TPMs inside just another container. This special vTPM management container can have access to the hardware based TPM and expose vTPM interface to the other containers through a communication channel. In practice, this channel can be local UNIX domain socket or another IPC mechanism.

The advantage of moving the implementation from kernel to the userspace comes from the arguably easier implementation process. Instead of kernel module, a daemon will process the requests from the other containers.

Depending on the exact nature of the IPC mechanism chosen for the TPM command transfer. The container using the service could need an additional piece of software that presents the IPC interface as a standard character device. In some ways, this way of implementing software TPMs is similar to the TPM simulator project by Strasser and

Sevnic et al. [24]. In their design they have a dedicated daemon providing the TPM functionality and an additional component exposing the daemon through the low-level device interface.

What differentiates our scenario from a mere simulator is the need for the daemon to be associated with the underlying trusted computing base. The simulator only provides the functionality but does not begin to offer the security guarantees associated with trusted platform modules.

However, moving vTPM implementation from the kernel to the user space – even if the daemon is running in a separate container – has still potential security implications. Trusted platform modules, regardless of their substrate, have to be able to protect their secrets. One of the key benefits of a hardware based TPM is that they are separate physical entities that can arguably protect keys from software based attacks in a more effective manner than pure software based solutions. Processes have many ways of inspecting and modifying each other's memory and controlling execution. Compared to this, placing the software TPM to kernel provides some protection, since the kernel can better limit this kind of access it exposes to the user space.

Container systems should isolate processes belonging to different containers [19]. In practice, this functionality is provided currently in Linux based container solutions through the use of resource namespaces. PID namespaces allow different containers to have entirely different view of processes running on the system. Along with other namespaces, they provide the fundamental building blocks of containers. If the system is working as it is supposed to, no process belonging to a different container can access software TPM's state in another container. In practice, there have been cases where this has not been the case, and container escapes are certainly not unheard of. Also, if an attacker manages to move from a container to the host, the attacker is likely able to gain access into other containers.

3) TPM with a native support for virtualization: In this paper, we have not explored the option of having a dedicated hardware TPM with a native support for virtualization, and we list the proposal here only for the sake of completness. There is existing research aiming to introduce direct support for virtualization at the hardware level [7]. If such device were to be implemented, we believe it would be likely to be usable in the context of containers, even if the original reasoning was framed to be used with the more traditional hypervisor-based systems.

4. CONCLUSION AND FUTURE WORK

In this work, we have provided an overview of the research concerning virtualized trusted platform modules. We have considered the challenges of applying existing approaches, originally designed to be used in a hypervisor-based setting, to a container-based system. Additionally we have provided two different architectural solutions for providing vTPM services for containers.

Future work is still needed to further specify the exact mechanisms for trust extensions from hardware based TPM to our proposed vTPM design. However, we believe that many of the existing approaches could be modified to work in a container-based environment. We are yet to implement the proposed architecture and doing so will be a chance for future research.

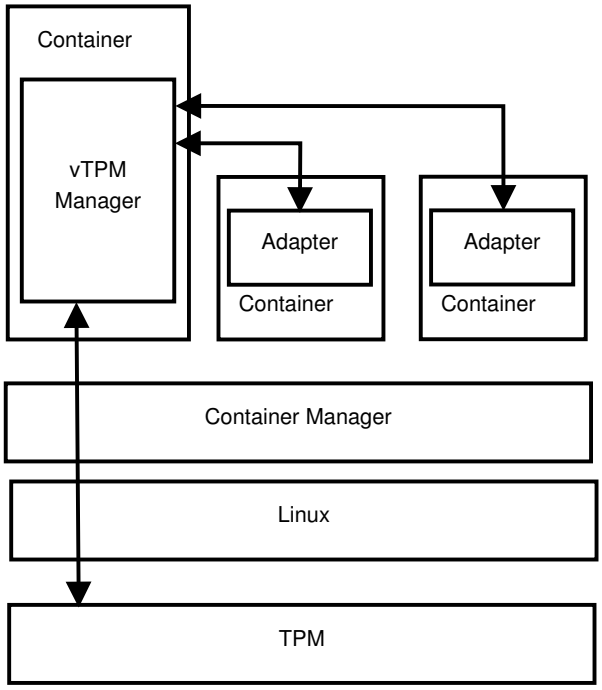


Figure 4: vTPM located in a dedicated container.

Acknowledgment

The authors gratefully acknowledge Tekes âĂŞ the Finnish Funding Agency for Innovation, DIMECC Oy, and the Cyber Trust research program for their support.

5. REFERENCES

[1] Cisco Application-Centric Infrastructure (ACI) and Linux Containers. http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/application-centric-infrastructure/white-paper-c11-732697.html. Accessed: 2016-08-05.

[2] TCG Specification Architecture Overview Specification Revision 1.4, 2007. http://www.trustedcomputinggroup.org/tcg-architecture-overview-version-1-4/. Accessed: 2016-08-08.

[3] Trusted Computing Group (TCG). http://www.trustedcomputinggroup.org/. Accessed: 2016-08-08.

[4] Trusted Platform Module (TPM). http://www.trustedcomputinggroup.org/work-groups/trusted-platform-module. Accessed: 2016-08-03.

[5] Virtual Trusted Platform Module- IBM Reseach. http://researcher.watson.ibm.com/researcher/view_group.php?id=2850. Accessed: 2016-08-08.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[7] S. Berger, R. CÃąceres, K. Goldman, R. Perez, R. Sailer, and L. Doorn. vTPM: Virtualizing the

trusted platform module. In *In USENIX Security*, pages 305–320, 2006.

[8] B. Danev, R. J. Masti, G. O. Karame, and S. Capkun. Enabling secure VM-vTPM migration in private clouds. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 187–196, New York, NY, USA, 2011. ACM.

[9] P. England and J. Loeser. *Para-Virtualized TPM Sharing*, pages 119–132. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[10] P. Fan, B. Zhao, Y. Shi, Z. Chen, and M. Ni. An improved vTPM-VM live migration protocol. *Wuhan University Journal of Natural Sciences*, 20(6):512–520, 2015.

[11] D. Fernandes, L. Soares, J. Gomes, M. Freire, and P. Inácio. Security issues in cloud environments: a survey. *International Journal of Information Security*, 13(2):113–170, 2014.

[12] K. Goldman and S. Berger. TPM main part 3 IBM commands, 2005.

[13] A. Grattafiori. Understanding and hardening Linux containers. Whitepaper, NCC Group, apr 2016.

[14] Z. Hong, W. Juan, and Z. HuanGuo. A trusted VM-vTPM live migration protocol in clouds. In *1st International Workshop on Cloud Computing and Information Security*. Atlantis Press, 2013.

[15] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–10, Aug 2009.

[16] X. Liang, R. Jiang, and H. Kong. Secure and reliable VM-vTPM migration in private cloud. In *Instrumentation and Measurement, Sensor Network and Automation (IMSNA), 2013 2nd International Symposium on*, pages 510–514, Dec 2013.

[17] J. Oberheide, E. Cooke, and F. Jahanian. Empirical exploitation of live virtual machine migration. In *Proc. of BlackHat DC convention*, 2008.

[18] G. Proudler, L. Chen, and C. Dalton. *Trusted Computing Platforms: TPM2.0 in Context*. Springer, Bristol, United Kingdom, 2014.

[19] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan. Security of OS-level virtualization technologies. In *Secure IT Systems:19th Nordic Conference, NordSec 2014, Tromsø, Norway, October 15-17, 2014, Proceedings*, pages 77–93. Springer, 2014.

[20] N. Santos, K. P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *HOTCLOUD*. USENIX, 2009.

[21] A. L. Shaw, B. Bordbar, J. Saxon, K. Harrison, and C. I. Dalton. Forensic virtual machines: Dynamic defence in the cloud via introspection. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 303–310, March 2014.

[22] Z. Shen and Q. Tong. The security of cloud computing system enabled by trusted computing technology. In *Signal Processing Systems (ICSPS), 2010 2nd International Conference on*, volume 2, pages V2–11–V2–15, July 2010.

[23] Y. Shi, B. Zhao, Z. Yu, and H. Zhang. A security-improved scheme for virtual TPM based on KVM. *Wuhan University Journal of Natural Sciences*, 20(6):505–511, 2015.

[24] M. Strasser and P. Sevnic. A software-based TPM emulator for Linux. *Semesterarbeit, ETH Zurich*, 2004.

[25] F. Stumpf and C. Eckert. Enhancing trusted platform modules with hardware-based virtualization techniques. In *2008 Second International Conference on Emerging Security Information, Systems and Technologies*, pages 1–9, Aug 2008.

[26] X. Wan, Z. Xiao, and Y. Ren. Building trust into cloud computing using virtualization of TPM. In *Fourth International Conference on Multimedia Information Networking and Security*, pages 59–63, Nov 2012.

Publication IV

Security in the Internet of Things through Obfuscation and Diversification

Shohreh Hosseinzadeh, Sampsa Rauti, Sami Hyrynsalmi, Ville Leppänen. In: International Conference on Computing, Communication and Security (ICCCS), 1–5. IEEE, 2015.

Security in the Internet of Things through Obfuscation and Diversification

Shohreh Hosseinzadeh, Sampsa Rauti, Sami Hyrynsalmi and Ville Leppänen
Department of Information Technology
University of Turku, Finland
Emails: {shohos, sjprau, sthyry, ville.leppanen} @utu.fi

***Abstract*—Internet of Things (IoT) is composed of heterogeneous embedded and wearable sensors and devices that collect and share information over the Internet. This may contain private information of the users. Thus, securing the information and preserving the privacy of the users are of paramount importance.**

In this paper we look into the possibility of applying the two techniques, obfuscation and diversification, in IoT. Diversification and obfuscation techniques are two outstanding security techniques used for proactively protecting the software and code. We propose obfuscating and diversifying the operating systems and APIs on the IoT devices, and also some communication protocols enabling the external use of IoT devices. We believe that the proposed ideas mitigate the risk of unknown zero-day attacks, large-scale attacks, and also the targeted attacks.

Keywords—Internet of Things, IoT, security, privacy, obfuscation, diversification

I. INTRODUCTION

Internet of Things (IoT) or Internet of Everything, was introduced by MIT Auto-ID Labs in 1999 [1], is a network of physical devices (objects) connecting to each other (wireless) for sending/receiving data. The tiny chips embedded in the devices enable them to communicate without the human interaction. This aims to make the human lives more intelligent, automated and thus more comfortable. IoT is known as the third revolution in information technology after the Internet and mobile communication networks, and today it is being used in multitude public and private sectors, ranging from the public safety to health care. By the continuous growing trend, more and more"things" are getting connected to each other every day, collecting and transmitting personal and business information back and forth. Cisco IBSG [2] reports that the number of connected devices in 2015 is 25 billion, and this number is expected to grow to 50 billion by 2020. However, the security in IoT is still a major challenge. According to [3], 70 percent of the IoT devices are vulnerable to exploits that could be a doorway for attackers to the network. On this basis, researchers and developers are continually seeking effective techniques that boost the security in this environment, which is compatible with the limitations of the participating nodes in IoT. To the best of our knowledge, none of the existing research works in the field of IoT security have looked into the obfuscation and diversification techniques as the potential techniques for mitigating the risk of malware.

In this paper, we propose a novel idea that addresses possible security threats in IoT. We base our idea on two promising techniques, obfuscation and diversification, that have been proved to be successful in impeding the malware in various domains [4]. In this work-in-progress paper, we propose using these two techniques to protect the operating systems and APIs of the devices participating in IoT, and also to introduce an additional level of security at the network level, by diversifying some protocols used in the communication. This study is a research proposal, in which we present our novel ideas. In the future works, we consider implementing these ideas, and demonstrate the effectiveness of the proposed approaches.

The remainder of the paper is structured as follows: in Section 2 we present a background on the characteristics of IoT, the software and the protocols used in these domains. Section 3 discusses our proposed idea in detail, and Section 4 concludes the paper.

II. CHARACTERISTICS OF THE IOT

Operating systems and software in IoT

IoT comprises a wide variety of heterogeneous sensors and devices, of which some are powered by more potent 32-bit processors (e.g., smart phones) and some are controlled by lightweight 8-bit micro-controllers [5]. Therefore, the chosen software should be applicable to a range of devices, including the lowest power ones. On one hand, it should be capable of supporting the functionality of the object; and on the other hand, should be in line with the limitations of these devices in memory, computational power, and energy capacity. The software in IoT should have the following characteristics [6]:

- **Heterogeneous hardware constraints:** the IoT software should have limited CPU and memory requirements, so that it could support the constrained hardware platforms.
- **Autonomy:** It should be energy efficient, reliable, and adaptive to the network stack.
- **Programmability:** It should provide a standard Application Program Interface (API) for software development and support the standard programming languages.

These factors have leaded the developers to think of the operating systems that are adaptive to the diverse low-power objects in IoT. Among all, Contiki [7] and TinyOS [8] are

978-1-4673-9354-6/15/$31.00 ©2015 IEEE

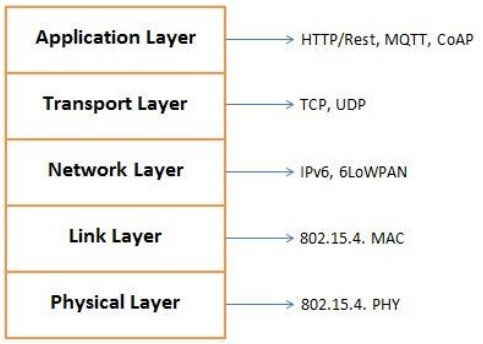


Figure 1. IoT network stack, and some examples of the corresponding protocols used in each layer [6].

the most dominant operating systems used on IoT devices. **Contiki** [7] is an open-source Operating System (OS) developed in C with a modular structure. It connects low-power micro-controllers to the Internet, and supports a wide range of networking standards, including IPV4, IPV6, CoAP. This lightweight OS is considered reasonably memory efficient by using only few kilobytes of memory[1]. Contiki is designed based on the event-driven multi-threading kernel. Moreover, Contiki has a dynamic nature, i.e., it allows the dynamic loading and unloading of applications at run-time. **TinyOS** [8] is an open source, application specific OS designed and widely used by low-power devices in WSN, and ubiquitous environments. It is implemented using the component-based programming model and supports the event-driven concurrency. TinyOS is multi-threading and has monolithic structure.

Access protocols in IoT

IoT encompasses a huge number of devices and applications connected to each other. Basically, there are three different types of connections in IoT [9]: a) Device to Device (D2D) is the connection between the devices, b) Device to Server (D2S) is the connection for sending collected data from the devices to the servers, and c) Server to Server (S2S) is the connection between the servers to share the collected data. For making the connections feasible there exist protocols that are designed based on the requirements and characteristics of the participants in IoT. As an example, Constrained Application Protocol (CoAP) [10] is an application layer protocol which is designed to be used in resource-constrained devices and WSN nodes. This protocol is simply translated to HTTP which simplifies the integration with the web. Figure 1 illustrates the current network stack used in IoT, and some of the communication protocols that are used in each layer.

1. Contiki: The Open Source OS for the Internet of Things: http://www.contiki-os.org

Security of IoT

IoT is still not a mature technology and accordingly, the security measures considered for it are still in early stages. Due to the fact that IoT is based on the Internet, it is subject to traditional security challenges threatening the Internet; and in addition to those traditional challenges, the key characteristics of IoT add new security threats on top of them.

These characteristics are [11], [12]: *Diversity:* The participating devices in IoT range from low power/cost to high performance devices. Hence, the security measure designed needs to be compatible with wide range of devices. *Scale:* The number of sensors and devices in IoT is tremendously growing, which makes it harder to control the devices and the collected information. *Wireless connection*: devices are connecting to the Internet wireless through different links (Bluetooth, 802.11, ZigBee). The links should be protected so that the communicated information does not leak. *Embedded use:* most of the IoT devices are designed for single purpose with different communication patterns. *Mobility:* The devices in IoT are mobile and connected to the Internet through various providers.

These properties make IoT more prone to the security threats compared to the Internet and traditional sensor networks. Security threats known in IoT could be due to insufficient security software or firmware, insufficient security in the Web interface, insecurity in cloud interface, insecurity in mobile interface, insecurity at authentication or authorization point, lack of proper security measures in network services, lack of encryption, poor security configuration, poor physical security, and privacy issues [12]. To overcome these security challenges, different tactics are suggested to be taken into account for implementation, including secure firewall and intrusion prevention system, secure identity management and access control program, multi-factor authentication, emergency response program, and IoT security standardizations [13]. However, we believe that they do not sufficiently secure the IoT and more concrete security measures are required. This motivated us to propose a novel idea to boost the security in IoT.

III. ENHANCING THE SECURITY OF IOT USING OBFUS- CATION AND DIVERSIFICATION TECHNIQUES

Background on obfuscation and diversification

By using the obfuscation and diversification techniques we do not aim at removing the security vulnerabilities, but making it challenging and costly for the intruder to take advantage of them.

Obfuscation refers to making the code more complicated and harder to understand. As a result, the attacker needs to spend more time and energy to comprehend the code [14]. Figure 2 illustrates how obfuscation affects the code: a) is a piece of JavaScript code, and b) is obfuscated version of the same code. As the figure shows, the code is

scrambled/obfuscated in a way that it is harder to understand the purpose of the code.

a)

```
function setText(data) {
  document.getElementById("myDiv").innerHTML
= data;
}
```

b)

```
function ghds3x(n) {
  h = "\x69\u006En\u0065r\x48T\u004DL";
  a="s c v o v d h e , n i";x=a.split("
");b="gztxleWentBsyf";
r=b.replace("z",x[7]).replace("x","E").replace("s
","").replace("f","I")
  ["repl" + "ace"]("W","m")+"d";
c="my"+String.fromCharCode(68)+x[10]+"v";
s=x[5]+x[3]+x[1]+"um"+x[7]+x[9]+"t";d=this[s][r](
c);if(+!![])
  { d[h]=n; } else { d[h]=c; }
}
```

Figure 2. a) Original version of a piece of JavaScript code, and b) obfuscated version of the same code

Diversification makes the program instances unique, so that the attacker is no longer able to exploit a vast number of devices via a single attack model. The attacker needs to design various versions of attack models intended for various program instances. This makes the attack costly and difficult. Figure 3 illustrates the program diversification, in which different replicas of program P are generated and distributed to the users. All these programs function the same, but have different internal structures. Therefore, even if the attacker succeeds in attacking one version of the program (P3), the other versions are safe.

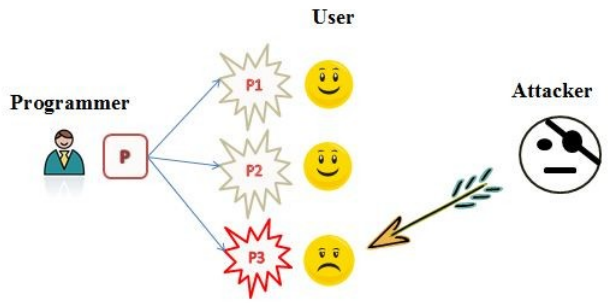


Figure 3. Diversification generates unique instances of the software. A single attack model works only for one instance.

The idea in diversification is to change all kinds of internal interfaces appearing inside IoT systems, or between IoT device and its backend (cloud) system. As diversification can be done with respect to known reference interfaces, software developers of IoT systems need not to be aware of diversification details. In practice, the purpose of diversifying an interface is to create a machine/system-wise unique secret that is shared only between the legal ”clients” of the interface. However, a diversified implementation is not the same as an encrypted implementation – as the idea is that a diversified implementation is still executable as such. Moreover, the idea behind the diversification techniques is not to remove the vulnerabilities and the security holes in software, but to prevent the attackers to take advantage of them, since even if an attacker would be able to use a security hole to inject malware (code) into the system, the malware would not work as it would need to access resources using the diversified (secret) interfaces instead of the prevailing setting where accessing only requires knowledge on reference interfaces. Therefore, diversification is considered very useful for securing the systems in the current era of OS monoculture and in the settings where it is hard to patch security problems of OS or system later on.

There exist different diversification techniques that use Various transformation mechanisms at different levels of software life cycle [16]. The idea of program diversification for effectively protecting the operating systems has been pioneered by Cohen [17]. In our previous works we have applied the diversification and obfuscation to different OSs, e.g., Linux to secure it against malware. As malware is meant to run on the user’s computer to manipulate the system or perform what an attacker desires [18], in order to thwart some malware to access resources via the system calls, we have to introduce a way to block the system calls made directly and indirectly from an untrusted application. For this purpose, we can diversify the system call numbers, and the library functions that make system calls [19], [20]. In [21] we diversified also the indirect entry points to the system calls and propagated the new information on entry points to all legal applications, so that the malware will not have the information to access the resources. On that account, we designed a concrete diversification tool which diversifies the symbols in the Linux ELF binaries and makes them unique.

The proposed approach

As mentioned, security threats in IoT could potentially be on the application layer (the software on the IoT devices) or on the network layer. On this basis, we propose two novel techniques to provide security in these two layers. We propose: 1) obfuscating/diversifying the OSs and APIs used in the IoT devices, and 2) obfuscating/diversifying some of the access protocols among them.

The sensors and devices in the IoT contain embedded chips which function with the help of an OS and APIs. This OS and the APIs are prone to malware. Our first proposed idea aims at protecting the IoT devices by applying diversification and obfuscation techniques to the OSs and APIs and securing them against the malware. As discussed earlier, in our previous works [19], [20], [21], we demonstrated that by obfuscating the OS and diversifying the APIs we can successfully make it harder for the malware to interact with the interfaces and access the resources.

We believe the same approaches can be applied on the OSs and APIs of the IoT devices to protect them from the malware. By making the OS and the APIs unique through diversification, we manage to thwart the massive-scale attacks, i.e., the attacker by designing a single attack model cannot take a large number of devices under control. Additionally, we mitigate the risk of zero-day attacks.
Secondly, we propose applying diversification at the access protocol level. In a communication network, an application level protocol specifies the interfaces and the shared protocols used by the communication parties. Protocol identification is the act of identifying what protocol is used in the communication session [22]. Protocol identification can be done via static analysis methods and comparing the

protocol used in the communication with the common existing protocols. The information gained from this analysis could be used by an intruder which endangers the integrity and confidentiality of the communication. In order to protect the protocol from identification, it could be obfuscated so that it is more difficult to be recognized by the traffic classification machines. Protocol obfuscation attempts to remove the properties that make the protocol identifiable, e.g., packet size and byte sequence (make them look random). The most common way for protocol obfuscation is using cryptography. Depending on the need of the network, different levels of encryption could be applied [22].

We propose obfuscating the communication protocol among a small set of nodes (e.g., within a home) in a way that the obfuscation method is kept secret among them and only the nodes which know the secret are able to communicate with each others. By changing/complicating the form of the protocol and making it different from the default format, we aim at generating a huge number of unique diversified protocols from a reference protocol. Applying diversification to the application is feasible, although it has two main challenges: 1) the recognition of transition (message sending/receiving) from programs, 2) complication and slowdown of the program. Besides the protocol obfuscation, we propose protocol diversification, which considers the protocol as an operation of two state machines, so that (synchronized) state changes are messages sent between parties. The original implicit state machine of a protocol can be diversified by adding/splitting new states and transitions.

Motivation

For securing IoT and the participants of these environments, we propose using diversification and obfuscation mechanisms. We believe that these two techniques are effective for this purpose, by mitigating the risk of unknown zero-day attacks and also preventing the large-scale and targeted attacks. The motivation behind our idea is as follows:

- Additional Security: There have been various security mechanisms designed and applied on IoT at the network level. We believe that presenting security at the device level is an orthogonal proactive measure for security. In this way, the malware is stopped at one node and not propagated to the whole network.
- Energy-efficiency: The participating nodes (devices) in IoT are extremely constraint in resources, i.e., they are limited in terms of memory, energy capacity, and computational power. Therefore, the security mechanism designed for them should be rather lightweight. For instance, the anti-virus programs often cannot be used in IoT devices (because of their huge effect on the performance and energy consumption). We believe that API diversification will not slow down the execution at all: however, obfuscation and protocol diversification affect the efficiency to some extent.
- No complexity for the manufacturer: The devices in IoT are controlled by the tiny chips embedded in them which may not tolerate a complex design. Considering this fact, our security mechanism does not add any complexity overhead for the manufacturer.
- Alleviate the risk of malware: The devices in IoT contain tiny chips with a lightweight OS on them. The OS handles the operation of the device by executing the code. Code execution presents a surface for attacks for unwanted software or malware. This malicious behavior could range from unauthorized accessing/reading of the data to altering it towards the attacker's intent. Thus, we believe the OS can be protected using obfuscation and diversification mechanisms, in order to make the malware ineffective. In order to make our approach applicable with the limited capabilities of the devices (in computational power) we limit the level of obfuscation and stick to the less complicated diversification methods, e.g., identifier renaming.
- Alleviate the risk of massive-scale attacks: Currently, there are billions of devices manufactured and distributed in a"monoculture" manner, i.e., they are designed and produced identical which makes them have the same layout and consequently, the same security vulnerabilities. By designing one single attack model, an attacker can simply invade a wide range of devices. We believe that diversification, by introducing"multiculturalism", in software production is a proper proactive security mechanism for the widely distributed environments including IoT. The idea of multiculturalism in the software production alleviates the risk of massive-scale attacks.
- Amend the update limitation in embedded devices: One of the drawbacks of the embedded devices is that they usually cannot be updated, i.e., in case a security hole is found they cannot update themselves to receive the security patch. Hence, we need to think of a solution that does not try to remove the security holes and vulnerabilities of the software, but to make it difficult for the attacker to exploit them. Diversification and obfuscation techniques are helpful to be prepared for the unknown attacks; because the general idea behind these techniques is not to remove the security holes but to avoid the attacker from taking advantage of them.

Limitations

Despite of the great benefits our proposed approach brings along, there might be some limitations for it, including cost and adaptability of the network. Obfuscation on one hand protects the software and applications from the malicious reverse engineering by making them hard to read; and on the other hand, it introduces costs in terms of execution overhead, memory consumption, and code size increase. Additionally, when obfuscation is applied on the protocol, it should be assured that the both parties in the communication session support the obfuscated protocol.

V. CONCLUSION AND FUTURE WORKS

In this paper we pointed out that due to the special characteristics of IoT, the security is more challenging compared to the traditional networks. Therefore, it requires security measures that are in line with the capacity of IoT devices.

Diversification and obfuscation techniques are the two promising security techniques that have been presented previously, and practitioners can already start to adapt these techniques into use to protect their IoT networks. In this

work-in-progress we proposed two approaches using the two techniques, obfuscation and diversification as well as request further work in this topic.

First, we propose applying the techniques on the operating systems and APIs of the IoT devices to make it harder to breach through the devices. This is a traditional way of utilizing the techniques and it is expected that this has a significant impact on improving the security of the devices.

Second, we propose applying these techniques on related application layer protocols. The idea is to increase the workload of a malicious attacker by using these techniques together with other techniques such as cryptography to secure the communication between the devices and servers.

Third, to the best of our knowledge, there are no thorough study of attack vectors of IoT networks. Further inquiries should focus on defining different attack vectors in order to prove security analyses tools to start working with countermeasures.

Finally, to extend this work-in-progress, our future work is towards diversifying an IoT operating system of a device, and obfuscating and diversifying an application level (internal) access protocol used between the IoT devices and its cloud based back-end system. For the device/application we have planned to use a health care sector IoT device based on Contiki as it is the most common operating system. The target of protocol obfuscation and diversification is the CoAP used in the same device.

ACKNOWLEDGEMENT

The authors gratefully acknowledge Tekes, the Finnish Funding Agency for Innovation, DIGILE Oy and Cyber Trust research program for their support.

REFERENCES

[1] I. Bose and R. Pal, "Auto-id: Managing anything, anywhere, anytime in the supply chain," Commun. ACM, 48(8), pp. 100–106, Aug. 2005.

[2] "The internet of things- how the next evolution of the internet is changing everything," verified 2015-07-08. [Online]. Available: https://www.cisco.com/web/about/ac79/docs/innov/IoTIBSG_0411FINAL.pdf

[3] "Internet of things research study - hp report," verified 2015-07-08. [Online]. Available: http://www8.hp.com/h20195/V2/GetPDF.aspx/4AA5-4759ENW.pdf

[4] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SOK: Auto- mated software diversity," in Security and Privacy (SP), May 2014, pp. 276–291.

[5] K. T. Nguyen, M. Laurent, and N. Oualha, "Survey on secure communication protocols for the Internet of Things," Ad Hoc Networks, vol. 32, pp. 17 – 31, 2015.

[6] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt, "Operating systems for the IoT–goals, challenges, and solutions," in WISG2013, January 2013.

[7] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in Local Computer Networks, 2004. 29th Annual IEEE International Conference on, Nov 2004, pp. 455–462.

[8] P. Levis S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "Tinyos: An operating system for sensor networks," in Ambient Intelligence. Springer Berlin Heidelberg, 2005, pp. 115–148.

[9] M. Corson, R. Laroia, J. Li, V. Park, T. Richardson, and G. Tsirtsis, "Toward proximity-aware internetworking," Wireless Communications, IEEE, vol. 17, no. 6, pp. 26–33, December 2010.

[10] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (CoAP)," in Internet Engineering Task Force (IETF), 2014.

[11] S. Babar, P. Mahalle, A. Stango, N. Prasad, and R. Prasad, "Proposed security model and threat taxonomy for the Internet of Things (IoT)," in Recent Trends in Network Security and Applications, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2010, vol. 89, pp. 420–429.

[12] R. Roman, J. Zhou, and J. Lopez, "On the features and challenges of security and privacy in distributed Internet of Things," Computer Networks, vol. 57, no. 10, pp. 2266 – 2279, 2013.

[13] S. Sicari, A. Rizzardi, L. Grieco, and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead," Computer Networks, vol. 76, no. 0, pp. 146 – 164, 2015.

[14] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[15] "The International Obfuscated C Code Contest: http://www.ioccc.org/years.html," verified 2015-07-16.

[16] P. Larsen, S. Brunthaler, and M. Franz, "Automatic software diversity," Security Privacy, IEEE, vol. 13, no. 2, pp. 30–37, Mar 2015.

[17] F. B. Cohen, "Operating System Protection through Program Evolution," Comput. Secur., vol. 12, no. 6, pp. 565–584, Oct. 1993.

[18] E. Skoudis, Malware: Fighting malicious code. Prentice Hall Professional, 2004.

[19] S. Rauti, J. Holvitie, and V. Leppänen, "Towards a diversification framework for operating system protection," ser. CompSysTech '14. New York, NY, USA: ACM, 2014, pp. 286–293.

[20] S. Rauti, S. Laur´en, S. Hosseinzadeh, J.-M. M¨akel¨a, S. Hyrynsalmi, and V. Leppänen, "Diversification of system calls in linux binaries," in InTrust2014. LNCS, to appear September 2015, p. 15 pages.

[21] S. Laur´en, P. M¨aki, S. Rauti, S. Hosseinzadeh, S. Hyrynsalmi, and V. Leppänen, "Symbol diversification of linux binaries," in WorldCIS-2014. Infonomics Society, 2014, pp. 74–79.

[22] E. Hjelmvik and W. John, "Breaking and improving protocol obfuscation," Chalmers University of Technology, Tech. Rep, vol. 123751, 2010.

Publication V

Interface diversification in IoT operating systems

Petteri Mäki, Sampsa Rauti, **Shohreh Hosseinzadeh**, Lauri Koivunen, Ville Leppänen. In Proceedings of the 9th International Conference on Utility and Cloud Computing, pp. 304-309. ACM, 2016.

2016 IEEE/ACM 9th International Conference on Utility and Cloud Computing

Interface Diversification in IoT Operating Systems

Petteri Mäki
Department of IT
University of Turku, Finland
mapema@utu.fi

Sampsa Rauti
Department of IT
University of Turku, Finland
sjprau@utu.fi

Shohreh Hosseinzadeh
Department of IT
University of Turku, Finland
shohos@utu.fi

Lauri Koivunen
Department of IT
University of Turku, Finland
lauri@koivunen.work

Ville Leppänen
Department of IT
University of Turku, Finland
ville.leppanen@utu.fi

ABSTRACT

With the advancement of Internet in Things (IoT) more and more "things" are connected to each other through the Internet. Due to the fact that the collected information may contain personal information of the users, it is very important to ensure the security of the devices in IoT.

Diversification is a promising technique that protects the software and devices from harmful attacks and malware by making interfaces unique in each separate system. In this paper we apply diversification on the interfaces of IoT operating systems. To this aim, we introduce the diversification in post-compilation and linking phase of the software life-cycle, by shuffling the order of the linked objects while preserving the semantics of the code. This approach successfully prevents malicious exploits from producing adverse effects in the system. Besides shuffling, we also apply library symbol diversification method, and construct needed support for it e.g. into the dynamic loading phase.

Besides studying and discussing memory layout shuffling and symbol diversification as a security measures for IoT operating systems, we provide practical implementations for these schemes for Thingsee OS and Raspbian operating systems and test these solutions to show the feasibility of diversification in IoT environments.

CCS Concepts

•**Security and privacy** → *Software security engineering;*

Keywords

software security, diversification, IoT

1. INTRODUCTION

Internet of Things (IoT) is a network consisting of physical devices that are connected to each other. These devices communicate with each other by sending and receiving data. The goal is to make the human lives more comfortable and automated. IoT is already being used in several public and private sectors, and application areas range from health care to industrial applications.

The number of connected devices is growing every day, as more and more devices get connected to Internet. Gartner estimates that there are over 6 billion connected "things" right now and the number will grow to 20 billion by 2020 [11].

However, security is still a huge problem in IoT operating systems. According to [4], 70 percent of IoT devices currently have vulnerabilities that may allow the attackers to infiltrate the systems. Therefore, novel effective techniques improving the security of IoT environment are needed. At the same time, the solutions have to be compatible with the limitations of the devices connected to IoT. In this work, we look into two different diversification techniques as a potential way to mitigate the risk of certain malicious attacks.

The contributions of this paper are as follows. We study and propose applying linker shuffling and ELF symbol diversification in IoT operating systems. We also provide practical implementations for these schemes for Thingsee OS which is based on NuttX [3, 1] and Raspbian [2] and test these solutions to prove the feasibility of diversification in IoT environment.

The remainder of the paper is organized as follows. In Section 2 we present an introduction to the terms and techniques related to this research. Section 3 shortly reviews the related works. In Section 4 we present our implementations of memory layout shuffling and symbol diversification for IoT operating systems in detail and demonstrate their feasibility. In Section 5 we discuss the limitations of the study. Conclusions and future work suggestions are given in Section 6.

2. BACKGROUND

Obfuscation [9] is the process of scrambling the program code while preserving its functionality and semantics. Through obfuscation the program becomes harder to read and understand, and therefore reverse engineering and compromising the software becomes more challenging.

Diversification [8], on the other hand, refers to altering the structure and internal interfaces of software, with the aim of generating unique instances of the software. Diversified but functionally equivalent versions of the software are distributed to users. Diversification is a technique to break

UCC 2016 Shanghai, China

http://dx.doi.org/10.1145/2996890.3007877

monoculture nature of software deployment and introduces multiculturalism. If a piece of malware succeeds in getting knowledge about one of these instances and run its malicious code, this exploit works only on that machine and should not be workable on other machines. This is because the interfaces are diversified differently on each separate system. Diversification can be achieved by using obfuscation techniques (such as renaming functions).

As a practical example of diversification, attacks making use of memory's known layout structure can be prevented by uniquely diversifying the memory layout [16]. This method is known as *Address Space Layout Randomization* (ASLR). It is used to thwart the buffer overflow attacks enabling arbitrary code execution. In ASLR, the address space locations of a process are randomly reshuffled. This usually covers the base of the executable and the stack, heap and library positions [6]. The attacker cannot jump to a particular location in the reordered memory. Here, memory can be seen as an *interface* that has been diversified so that the attacker cannot use it anymore. In this paper, we present a link-time implementation of this shuffling scheme for Thingsee OS (on a Thingsee One device). We modified the GNU linker to introduce diversity into the resulting program after compilation and linking by shuffling the order of sections (functions and data objects) in the linked binary without changing the program semantics.

A *linker* is a program used in the build process of software that combines object files into a finished executable file or shared library, resolving dependencies between the object files and updating necessary positions of the object code with the correct relocation information. It therefore makes sense to apply diversification at this phase of software development/deployment.

Another example of diversification and a scheme we implemented for Raspbian (on a Raspberry Pi device) is symbol diversification. To use the critical resources of a device, applications often use well-known libraries. If we diversify the symbol names in these libraries, the adversary can no longer use the well-known operating system libraries to attack the system. This diversification is propagated accordingly to the trusted programs. As a result of this, the diversified applications know the diversification secret (the diversified symbol names) and are compatible with the library binaries containing important functions. Besides library functions, some IoT operating systems also support system calls to provide services for user programs – in such settings, system call diversification should be applied together with symbol diversification.

As a security measure, diversification is orthogonal, meaning it can be used with combination with other security measures like encryption and therefore brings additional security to a system. Diversification is also transparent in the sense it does not affect the user experience or cause additional work for the programmer.

3. RELATED WORK

There has been a large body of research on using diversification for securing software through many different diversification methods with the aim of mitigating several different security attacks [17]. In a previous study, we presented a survey on various kind of diversification methods available [14].

System call diversification [22, 18] is a diversification technique for protecting the operating systems from harmful malware. Laurén et al. [18] have proposed changing the system call numbers in the kernel and also the applications that invoke those system calls. Therefore, a malware that does not have the knowledge about new system call interfaces cannot interact with the environment and becomes ineffective. Another work done by Rauti et al. [22] proposes diversifying the system calls in ELF binaries.

Symbol diversification is another technique that diversifies the symbol names that are used with dynamic linking of shared libraries. A shared library implements some functionality and provides that for use in executables and other shared libraries by associating the functionality with symbol names that can be used elsewhere to find the functionality. Symbol diversification has already been experimented with under x86_64 Linux. [19]

As discussed earlier, in the literature there exist many different diversification techniques that each apply diversification at various stages of software development life-cycle, for instance design, implementation, compilation, post-compilation (e.g. linking), distribution, and execution. Among all, compile-time and link-time are most preferred stages for applying diversification. Compile-time diversification is a fairly automatic process where the compiler creates unique binaries each time [10]. Class transformation for Java byte-code diversification [20] is an example for post-compile time diversification. In [7] diversification happens after compilation, during linking and before execution. In this work, the location of program code and data is obfuscated and randomized in a way that makes it more difficult for a malicious program to find the addresses for program segments, and modify the program.

Hosseinzadeh et al. [12, 13] have proposed using diversification for IoT networks as a means to boost the security of the network and also the participating devices in this network. In these works, the authors propose diversifying the operating system interfaces of the IoT devices (security at application layer). The rationale behind this idea is to thwart large-scale attacks. Second they propose diversifying some of the communication protocols and providing security at the network level.

4. DIVERSIFYING THE MEMORY LAYOUT AND SYMBOLS IN IOT OPERATING SYSTEMS

We constructed a modified linker for diversifying the memory layout of Thingsee OS and demonstrated how this can prevent an exploit attempt from working on Thingsee one device (Section 4.2). Additionally, we applied and tested symbol diversification in Raspbian ELF files (Section 4.3).

We apply the proposed diversification methods in practice into an existing device and measure the effectiveness of this approach by first testing them against a working exploit and gauging the feasibility and effectiveness of the methods when thwarting malicious attacks.

4.1 Introducing layout shuffling to GNU linker

Our modified version of GNU ld linker was created with very few modifications to the original source code. To obtain the desired functionality, three files have to be changed, with about 100 added and one or two deleted or modified source code lines. GNU ld already contains functionality to sort

sections whose names match a wildcard expression defined in a linker script by name or alignment. We introduced an additional way of sorting that sorts the wildcard-matching sections by salted md5 hashes of their names. There should be no need to use the same salt more than once, since the memory layout itself is rarely an interface that is needed by legal code, other than the linked program itself which has the appropriate addresses provided by the linker. Our scheme might not be the optimal way of diversification, but it has worked well in our experiments. To increase diversity in the linked software, the source code should be compiled to object files in a way that every data object and function gets its own section. With *gcc*, this means passing `-ffunction-sections` and `-fdata-sections` as parameters for the compiler.

4.2 Layout shuffling of Thingsee OS on Thingsee One

Many resource-constrained and often time-critical IoT devices and operating systems do not support many ordinary security measures, such as virtual memory and no-execute bits, that are commonly found in modern desktop and server computing platforms [21]. This is also the case with the default "flat" NuttX-based Thingsee OS build. If there is no memory protection, specialized system call traps, or dynamic shared libraries, the prospects for different diversification schemes are somewhat more limited.

We decided to apply layout diversification to Thingsee OS with a flat address space configuration. The aim was to prevent return-oriented programming (ROP) attacks [23] depending on certain data residing at known addresses from functioning. First, we wrote a vulnerable Thingsee OS application program that writes user-supplied data to a stack buffer without bounds-checking. We devised an exploit that overwrites a return address on the stack and makes the program jump to a chosen function. After executing the function, the system apparently crashes and reboots, preventing the execution of malicious code. Later, we rebuilt Thingsee OS using our modified `ld` with randomization of the order of functions and data in the Thingsee OS image. An identical exploit did not make the chosen function to be executed this time, but it nevertheless invoked undefined behavior and, in our case, made the OS crash again.

Our vulnerable program was simply created by modifying the "Hello World" example in Thingsee OS in order to facilitate creating a new Thingsee OS application. Our vulnerable program takes an input from a command-line argument as a hexadecimal string and writes it to a stack buffer in a hex-decoded form without bounds-checking. We use this method of input to simulate a malicious input that would be in a real application environment, such as a network server.

We compiled Thingsee OS on Lubuntu 16.04 x86_64 using arm-none-eabi-gcc version 4.9.3 and our modified versions of binutils-gdb (git commit `6e2565079204ae2d2c0a5fa15fcd233e9c614f0b`), and thingsee-sdk (git commit `b65cfa8ec466d498e24959b90568b076c942aa6d`).

We built the undiversified version of Thingsee OS as follows:

```
cd thingsee-sdk/nuttx/tools/; ./configure.sh haltian-
    tsone/nsh
# Enable the "Hello, World!" example
cd ..; make menuconfig
make
arm-none-eabi-objdump -d nuttx | less
# Find 'hello_main' (0x08014174)
# and the address of 'evil' (0x08013fcc).
# May be different, depending on versions and
    configuration
# Set the device to flash mode and upload the OS:
sudo dfu-util -d 0483:df11 -a0 -D nuttx.dfu -s :leave
socat - /dev/ttyACM0,rawer
```

The following listing shows the Thingsee OS command line shell. First, we examined the stack to find where the return address resides. We then devised and ran the exploit. The ARM processor needs the return address be incremented by one when using Thumb code. [5] Thus, we add 1 to the address of 'evil' and write it to the stack in little-endian byte order.

```
NuttShell (NSH)
nsh> hello 42 0 20
hello 42 0 20
hex_strcpy: a=20009030; s_d=0; s_n=32
00000000000000000000000000000064900020
    b5410108754101080000000
foo 1
B
nsh> hello 10112233445566778899aabbccddeeff11111111
    cd3f0108
hello 10112233445566778899aabbccddeeff11111111cd3f0108
hex_strcpy: a=20009010; s_d=0; s_n=32
00000000000000000000000000000044900020
    b54101087541010800000000
foo 1
(some unintelligible characters)
Evil function!
```

After printing "Evil function!", the USB connection to our device breaks as the system crashes.

The layout-shuffled version of Thingsee OS was built as follows:

```
make distclean && cd configs/haltian-tsone/
patch Make.defs <<END_OF_PATCH
79,80c79
< LDFLAGS += --gc-sections
< # --sort-section=shuffle_obfuscation
---
> LDFLAGS += --gc-sections --sort-section=
    shuffle_obfuscation
END_OF_PATCH
cd ../../tools/  &&  ./configure.sh haltian-tsone/nsh
make menuconfig
# find and select "Hello, World!" example,
# exit until it prompts to save, save
env SHUFFLE_OBFUSCATION_SALT=foo make
# set Thingsee to flash mode
sudo dfu-util -d 0483:df11 -a0 -D nuttx.dfu -s :leave
socat - /dev/ttyACM0,rawer
```

Then the same exploit was run again:

```
NuttShell (NSH)
nsh> hello 10112233445566778899
    aabbccddeeff11111111cd3f0108
hello 10112233445566778899aabbccddeeff11111111cd3f0108
hex_strcpy: a=20009030; s_d=0; s_n=0

foo 1
(some unintelligible characters)
```

Again, the connection is broken. However, now the function 'evil' is not called. Undefined behavior is invoked nevertheless, and Thingsee OS crashes. In this particular experiment, execution happens to jump to halfway in the middle of a 4-byte instruction word that is part of a tiny piece of code that jumps to an exception handler. Figure 1 shows the execution in both of the cases described above.

A crude estimate of the granularity of Thingsee OS layout shuffling can be seen in Figure 4.2. It shows a histogram representing the distribution of bytes in differently sized *PROGBITS* sections in the source object files of Thingsee OS. 6.2

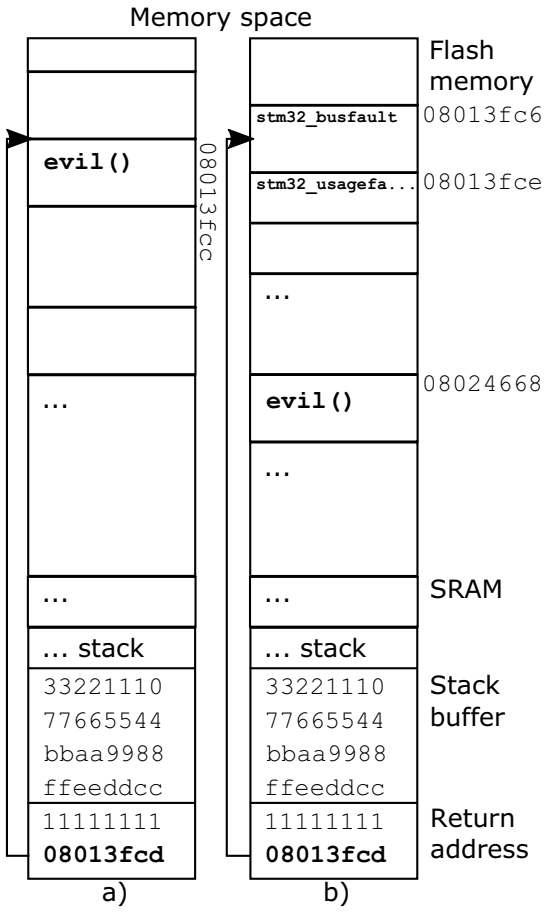


Figure 1: a) Normal OS: execution jumps to beginning of evil(), b) OS layout-diversification: in this particular case, execution jumps to the middle of an instruction.

MB out of 20 MB in total are found in sections at most 1000 bytes in size.

4.3 Symbol diversification on Raspbian

We used a symbol diversification tool that has previously been used to diversify x86_64 Linux [19]. The tool needed to be modified to support 32-bit ELF files, which was straightforward. The necessary source code for the symbol diversification tools in this experiment is comprised of approximately 2500 lines of C++ code and 300 lines of Python and Bash scripts. We also needed to modify the *glibc* dynamic linker source code with 225 inserted and 7 deleted lines in 6 files, plus an existing third-party SHA-2 implementation that consisted of 2 files and 1262 lines of code.

We managed to get Raspbian to start *systemd*. Most of the services seemed to start, but initially some did not. For example, the login service did not start, and thus we could not use the system. The problems were caused by run-time dynamic loading. For example, */bin/login* uses Pluggable Authentication Modules and *libdl*.

To overcome these challenges, we implemented a source code patch to *glibc* that makes it diversify symbols passed

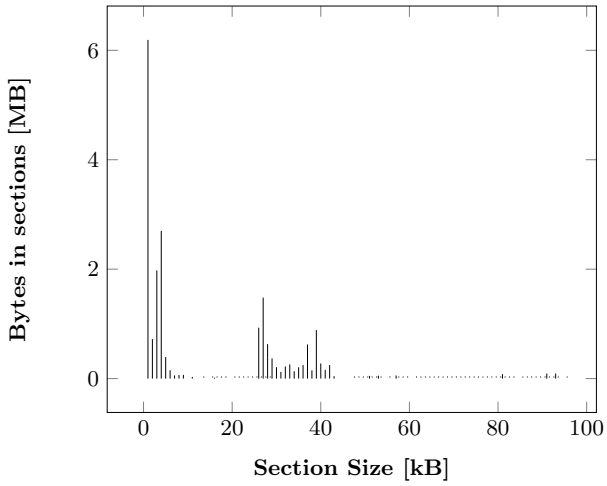


Figure 2: Histogram of the distribution of bytes in differently sized *PROGBITS* sections in the source object files of Thingsee OS.

to `dlsym` on the fly. The patched *glibc* contains a static data object that contains the diversification secret and the identification number of the algorithm to use. This data can then be rewritten to include the correct diversification secret in the library after compilation and linking, so including the secret is a very inexpensive operation.

With the patched glibc, a symbol-diversified Raspbian can boot, although only if using simple prefixation instead of salted SHA hashing as the diversification transform. `login` works, as does the shell basically, too. The programs `readelf`, `less`, and `ssh` work. Initially we had problems with `sudo`, but we diversified more types of symbols in the system to make it work. The `ping` command works after we set the set-user-id bit to its metadata. Our diversification tools did not copy that information. The `man` program cannot find a library it needs without manually specifying the location of the library with, for example, the *LD_LIBRARY_PATH* environment variable. `gdb` starts but has some problems that likely make it unusable for real debugging.

5. LIMITATIONS

Unfortunately, layout shuffling, especially in systems without memory protection, does not prevent damage or denial-of-service attacks caused by attempted security exploits. For devices with a small address space, brute-force attacks are also a potential problem. For example, the Thingsee One device has only 512 KB of flash memory for program code and in addition the device reboots when the system crashes, so brute-force testing of different addresses could be a problem without additional measures when given enough time. As an example, if one reboot takes 10 seconds, brute-force attacks should succeed within about 60 days. However, it is possible to set up other security measures to detect such attacks.

It is worth considering how fast compromising the system would be without the security measure. This would only take an instant. Ideally, in the case of computer worms and with diversification, a large scale compromise would not be

speed-limited only by network, but it would also be constrained by having to spend time and processing time on compromising more devices with a similar vulnerability.

Although our shuffling scheme does not entirely prevent all advanced return-oriented programming attacks. As the embedded device we used was not configured to use memory protection hardware, stack buffer overflow attacks could still utilize executable stack space, at least unless the stack addresses are randomized. Even if attackers cannot predict the address of the stack, they can utilize NOP slides to increase the probability of successfully having their malicious code executed [15]. Therefore, layout shuffling alone, without memory protection, will not prevent stack smashing attacks from unwanted code execution. Still, our scheme prevents several possible ROP attacks and would also considerably slow down the spread of internet worms from one device to another, for instance.

The limitations of symbol diversification are more implementation specific. Systems that are specifically designed with changing the symbol names in mind should be simple to diversify. Diversification can make binary-related activities, such as debugging, more difficult [17]. Naturally, more challenges emerge when adding symbol diversification to a system that does not support changing symbol names well.

Run-time dynamic loading presents one of the greatest challenges for symbol diversification. In POSIX systems, this is done using the `dlopen`/`dlsym` class of functions. Since the binary and library files in a diversified systems do not have access to the information about the original symbol names through their symbol tables, resources cannot be loaded from them by searching with the original name. Thus, the `dlsym` implementation needs to obtain the diversified name, either by getting the diversified symbol as a parameter from the its caller, or by computing the diversified symbol from an undiversified symbol name parameter. We consider the former approach more secure but also more challenging to implement, since all programs using dynamic loading should be analyzed and modified to provide the diversified versions of symbols.

Different types of symbols also pose a challenge. Some types of symbols may not be good for diversification. For example, there are symbol types such as STT_SECTION and STT_FILE in ELF binary format. While, for example, file names are one possible interface for diversification, it might not be a good idea to diversify STT_FILE symbols the same way as function and data objects. And if not all symbols are diversified, there might be a need to create separate diversifying and non-diversifying versions of `dlsym`, the programming interface to dynamic linking loader. Though the preferred method of handling `dlsym` would be to supply the diversified symbols in the application code that calls `dlsym`.

Additionally, the symbol diversifier that we used in our experiments complicates the structure of the ELF files, leaving zero-filled gaps into the file and moving data to unusual positions. The Linux ELF loader also has a quirk requiring the program header table to reside at a specific offset in the ELF file in relation to its address in the process image, for which we devised a workaround that may increase the file size. In our experiments, the executables and shared libraries, originally taking approximately 230 MB of space, increased their size by about 80% on average. Since much of the additional data is *NUL* bytes, using file systems that support sparse files will mitigate the actual effect of this additional space usage.

We have seen that there are some challenges for memory layout shuffling and symbol diversification in IoT operating systems. However, we have also shown that most of these challenges can be overcome or their effects can be reasonably mitigated.

6. CONCLUSION AND FUTURE WORK

We have presented two diversification approaches for IoT operating systems, memory layout shuffling and symbol diversification. We have also built and experimented practical implementations for these schemes to demonstrate their feasibility.

According to our observations, the layout-shuffled programs worked correctly in normal well-behaved cases. Undefined behavior, such as the stack smashing exploit, correctly causes the behavior of diversified programs to diverge from that of undiversified programs so that the adversary cannot invoke malicious functionality.

Layout shuffling scheme we devised can be applied to practically all computing devices, since its basic requirements are just an instruction memory and a processor that supports jumping to specified addresses there. Systems that already implement address space layout randomization (ASLR) do not benefit from this link-time layout shuffling, except for code to which ASLR is not applied.

It is best to apply the layout shuffling scheme we implemented at compile and link time, as the compiler should be informed to dedicate an individual section to each function and data object to maximize entropy, and the linker stage actually applies the layout randomization. Software vendors who do not want to disclose their source code could distribute their software as compiled object code, with possibly diversified symbol and section names, to be linked by the recipient of the software.

We also presented a scheme for diversifying the symbolic names of all the library entry points that lead to critical resources of a operating system. Our symbol diversification tool has been shown to be able successfully diversify executables in an IoT operating system so that their run-time functionality is not greatly affected, while preventing malicious attacks at the same time.

In practice, both presented approaches can be applied during the life cycle of a device by adding an extra processing stage before uploading firmware to each device. Each device gets an unique firmware modified with layout shuffling and/or symbol diversification before shipping. In a similar way, updating a device would work by giving each customer an unique instance of the updated code from a pregenerated buffer of diversified firmware binaries. This would also essentially switch diversification secrets on the device to fresh ones.

As future work, a more comprehensive diversification scheme covering several approaches could be tested in an IoT operating system. For example, system call diversification [18], which we did not include in our practical implementation in this paper, would also potentially be a useful technique in IoT environment.

Based on our experiments presented in this paper, we believe both memory layout shuffling and symbol diversification can be successfully applied to protect IoT operating systems and their applications from malicious attacks.

Acknowledgment

The authors gratefully acknowledge Tekes – the Finnish Funding Agency for Innovation, DIMECC Oy and Cyber Trust research program for their support. This research is also supported by Tekes project 1881/31/2016 "Cybersecurity by Software Diversification".

7. REFERENCES

[1] NuttX Real-Time Operating System. http://nuttx.org/. Accessed 17 August 2016.

[2] Raspbian. https://www.raspbian.org/.

[3] ThingSee One. https://thingsee.com/. Accessed 17 August 2016.

[4] Internet of things research study: 2015 report. http://www8.hp.com/h20195/V2/GetPDF.aspx/4AA5-4759ENW.pdf, 2015.

[5] Procedure Call Standard for the ARM® Architecture. http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf, 2015.

[6] M. Abadi and G. D. Plotkin. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.*, 15(2):8:1–8:29, July 2012.

[7] E. Bhatkar, D.C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[8] F.B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565 – 584, 1993.

[9] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscation Transformations. Technical Report 148, The University of Auckland, 1997.

[10] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.

[11] Gartner. Gartner Says 6.4 Billion Connected things Will Be in Use in 2016, Up 30 Percent From 2015. http://www.vxdev.com/docs/vx55man/vxworks/guide/c-vm.html. Accessed: 2016-06-23.

[12] S. Hosseinzadeh, S. Hyrynsalmi, and V. Leppänen. Obfuscation and diversification for securing the Internet of Things (IoT). In Rajkumar Buyya and Amir Vahid Dastjerdi, editors, *Internet of Things Principles and Paradigms*, chapter 14, pages 259–274. Morgan Kaufmann is an imprint of Elsevier, Cambridge, MA 02139, USA, 2016.

[13] S. Hosseinzadeh, S. Rauti, S. Hyrynsalmi, and V. Leppänen. Security in the internet of things through obfuscation and diversification. In *Computing, Communication and Security (ICCCS), 2015 International Conference on*, pages 1–5, Dec 2015.

[14] S. Hosseinzadeh, S. Rauti, S. Laurén, J.-M. Mäkelä, J. Holvitie, S. Hyrynsalmi, and V. Leppänen. A survey on aims and environments of diversification and obfuscationin software security. In *International Conference on Computer Systems and Technologies -CompSysTech'16*, page 8 pages, 2016. accepted-to be published in 2016.

[15] F.-H. Hsu, C.-H. Huang, C.-H. Hsu, C.-W. Ou, L.-H. Chen, and P.-C. Chiu. Hsp: A solution against heap sprays. *Journal of Systems and Software*, 83(11):2227 – 2236, 2010. Interplay between Usability Evaluation and Software Development.

[16] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 339–348, Dec 2006.

[17] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 276–291, May 2014.

[18] S. Lauren, S. Rauti, and V. Leppänen. Diversification of system calls in linux kernel. In Boris Rachev and Angel Smrikarov, editors, *Proceedings of the 16th International Conference on Computer Systems and Technologies*, volume 1008 of *ACM ICPS*, page 284–291. ACM Press, 2015.

[19] S. Laurén, P. Mäki, S. Rauti, S. Hosseinzadeh, S. Hyrynsalmi, and V. Leppänen. Symbol diversification of linux binaries. In C.A. Shonigun and G.A. Akmayeva, editors, *Proceedings of World Congress on Internet Security (WorldCIS-2014)*, page 75–80. Infonomics Society, 2014.

[20] Y. Le and H. Huo-Jiao. Research on java bytecode parse and obfuscate tool. In *International Conference on Computer Science Service System (CSSS)*, pages 50–53, Aug 2012.

[21] A. R. Pop et al. DEP/ASLR implementation progress in popular third-party windows applications. 2010.

[22] S. Rauti, S. Laurén, S. Hosseinzadeh, J.-M. Mäkelä, S. Hyrynsalmi, and V. Leppänen. Diversification of system calls in linux binaries. In Moti Yung, Liehuang Zhu, and Yanjiang Yang, editors, *Trusted Systems — 6th International Conference, INTRUST 2014*, Lecture Notes in Computer Science, page 15–35. Beijing Institute of Technology, 2014.

[23] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

Publication VI

Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization

Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, Andrew Paverd. In Proceedings of the 3rd Workshop on System Software for Trusted Execution (SysTEX '18), pages 42-47. ACM, 2018.

Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization

Shohreh Hosseinzadeh*
University of Turku, Finland
shohos@utu.fi

Hans Liljestrand*
Aalto University, Finland
hans.liljestrand@aalto.fi

Ville Leppänen
University of Turku, Finland
ville.leppanen@utu.fi

Andrew Paverd
Aalto University, Finland
andrew.paverd@ieee.org

ABSTRACT

Intel Software Guard Extensions (SGX) is a promising hardware-based technology for protecting sensitive computation from potentially compromised system software. However, recent research has shown that SGX is vulnerable to *branch-shadowing* – a side channel attack that leaks the fine-grained (branch granularity) control flow of an *enclave* (SGX protected code), potentially revealing sensitive data to the attacker. The previously-proposed defense mechanism, called *Zigzagger*, attempted to hide the control flow, but has been shown to be ineffective if the attacker can single-step through the enclave using the recent SGX-Step framework.

Taking into account these stronger attacker capabilities, we propose a new defense against branch-shadowing, based on control flow randomization. Our scheme is inspired by Zigzagger, but provides quantifiable security guarantees with respect to a tunable security parameter. Specifically, we eliminate conditional branches and hide the targets of unconditional branches using a combination of compile-time modifications and run-time code randomization. We evaluated the performance of our approach using ten benchmarks from SGX-Nbench. Although we considered the worst-case scenario (whole program instrumentation), our results show that, on average, our approach results in less than 18% performance loss and less than 1.2 times code size increase.

KEYWORDS

Intel SGX; side-channel attack; branch-shadowing attack

ACM Reference Format:
Shohreh Hosseinzadeh, Hans Liljestrand*, Ville Leppänen, and Andrew Paverd. 2018. Mitigating Branch-Shadowing Attacks on Intel SGX, using Control Flow Randomization. In *3rd Workshop on System Software for Trusted Execution (SysTEX '18), October 15, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3268935.3268940

*S. Hosseinzadeh and H. Liljestrand contributed equally to this work, which was done while S. Hosseinzadeh was visiting the Secure Systems Group at Aalto University.

SysTEX '18, October 15, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5998-6/18/10…$15.00
https://doi.org/10.1145/3268935.3268940

1 INTRODUCTION

Intel Software Guard Extension (SGX)[1] is a recent hardware-based Trusted Execution Environment (TEE) providing isolated execution and guaranteeing the integrity and confidentiality of data within an *enclave*. The enclave is protected from all other software on the platform, including potentially malicious system software (e.g., operating system, hypervisor, and BIOS). Additionally, SGX enables hardware-based measurement and attestation of enclave code.

Although Intel has stated that side-channel attacks are beyond the scope of SGX[2], recent research has demonstrated that SGX is susceptible to several side-channel attacks, which could leak secret information. In particular, Lee et al. [10] demonstrated a *branch-shadowing* side channel attack that allows untrusted software to learn the precise control flow of code running inside an enclave. If this control flow depends on any secret information, this side channel would leak the secret information. This attack abuses the CPU's Branch Prediction Unit (BPU), which is used to improve performance by allowing pipelining of instructions before exact branching decisions are known, i.e., whether or not branches are taken, and the targets of indirect branches. The BPU bases its decisions on recent branch history, which is stored in the CPU's internal Branch Target Buffer (BTB). Two critical factors allow this attack to proceed: 1) BTB entries created by branches inside the enclave are not cleared when the enclave exits; and 2) BTB entries only contain the lower 31 bits of the branch instruction's address, allowing the attacker to create *shadow* branch instructions outside the enclave that map to the same BTB entries as the enclave's branches. The attacker executes the victim enclave, interrupts it immediately after the branch instruction, executes the shadow branch code, and checks whether the branches were correctly predicted, thus revealing whether the BTB entry had been created by the enclave.

Lee et al. [10] also proposed a software-based defense against branch-shadowing, called *Zigzagger*. Using compile-time instrumentation, Zigzagger converts all conditional and unconditional branches into unconditional branches targeting Zigzagger's trampolines, i.e., minimal code sections that hold intermediate jumps — bounces — to the target locations. The Zigzagger trampolines initiate a series of jumps back-and-forth to different branches. The idea is that the attacker cannot interrupt the enclave with sufficient precision to shadow the target branch in this rapid series of jumps. However, SGX-Step [15] invalidates this assumption by

[1] https://software.intel.com/en-us/sgx
[2] https://software.intel.com/en-us/articles/intel-sgx-and-side-channels

showing how an enclave can be interrupted with single instruction granularity, thus breaking the Zigzagger defense.

The recent Spectre [9] attacks, and their subsequent SGX-specific SGXPectre variant [4] are similar to branch-shadowing in that they exploit the BPU. However, we have confirmed experimentally that neither recent firmware patches, nor the Retpoline compiler-based mitigation affect the ability to perform branch-shadowing attacks.

To overcome this challenge, we present a new defense against branch-shadowing, even if the attacker can single-step through the enclave. Similar to Zigzagger, we use compile-time modifications to convert all branch instructions into unconditional branches targeting our in-enclave trampoline code. At run-time, we then randomize the layout of our trampoline, forcing the attacker to shadow all possible locations. The finite size of the BTB limits the number of guesses the attacker can perform, and thus we can quantify and limit the success probability of a branch-showing attack using the size of the trampoline as a tunable security parameter.

Our contributions are therefore:

- Experimental analysis demonstrating that the recent Spectre mitigation techniques *do not* affect the branch-shadowing attack (Section 3).
- A new approach for defending against branch-shadowing attacks, even in the presence of single-step enclave execution, using control flow randomization (Section 5).
- An initial LLVM implementation of our solution (Section 6)[3] and a quantitative evaluation of its performance and security guarantees (Section 7).

2 BACKGROUND

2.1 Branch Prediction

Intel CPUs use instruction pipelining to load and execute instructions in batches. This allows optimization such as parallelizing and reordering of instructions. The CPU also performs speculative execution, i.e., it uses the BPU to predict which branches will be taken, and executes them before knowing if they are taken.[4] In modern microprocessors, the BPU typically consists of two main subsystems, a BTB and a directional predictor.

The BTB is used to predict the targets of indirect branches.[5] Whenever a branch is taken, a new record is created in the BTB associating the branch instruction's addresses with the target address. Upon encountering subsequent branch instructions, the BPU checks the BTB for the branch instruction address and, if an entry exists, it predicts that the current branch instruction will behave in the same way. The exact details of the BTB lookup algorithms, hashing and size are not public, but the BTB size on Intel Skylake CPUs has been experimentally determined to be 4096 entries [10]. The directional predictor is used to predict whether or not a conditional branch will be taken [5].

Multiple processes executing on the same core share the same BPU, allowing an attacker to misuse the BPU across processes to infer the target and direction of branch instructions [5, 10].

[3] Available online at https://github.com/SSGAalto/sgx-branch-shadowing-mitigation
[4] https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-optimization-reference-manual
[5] https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

2.2 Intel SGX

Intel SGX is an instruction set extension that provides new instructions to instantiate Trusted Execution Environments (TEEs), called *enclaves*, consisting of code and data. An enclave's data can only be accessed by code running within the enclave, thus protecting it from all other software on the platform, including privileged system software such as the OS or hypervisor. Enclave data is automatically encrypted before it leaves the CPU boundary. However, the OS remains in control of process scheduling and memory mapping, and can therefore control the mapping of (encrypted) enclave memory pages and interrupt enclave execution.

2.3 Branch-shadowing Attacks on SGX

In the *branch-shadowing* attack by Lee et al. [10], the attacker first statically analyzes the unencrypted enclave code and enumerates all branches (i.e., conditional, unconditional, and indirect) together with their target addresses. She then creates shadow code where the branch-instructions and target addresses are aligned such that they will use the same BPU history entries. The attacker then allows the enclave to execute briefly before interrupting it. Finally, she enables the performance counter, in particular the Last Branch Record (LBR), and executes the shadow code, prompting the CPU to predict shadow-branch behavior based on prior enclave execution. The LBR contains information on branch prediction but cannot record in-enclave branches. However, the in-enclave branches can be inferred from the LBR entries for the branches executed after exiting the enclave. Unlike cache-based channels, this does not require timing because the LBR directly reports prediction status.

2.4 Zigzagger and SGX-Step

Zigzagger [10] is presented as a software-based countermeasure to thwart branch-shadowing attack. Zigzagger removes the branches from the enclave functions by obfuscating and replacing a set of branch instructions with a series of indirect jumps. Instead of each conditional branching instruction, an indirect jump and a conditional move (CMOV) is used. Zigzagger assumes that an attacker cannot precisely time the enclave interrupts, i.e., a single probe will cover over 50 instructions. It introduces a trampoline to exercise all unconditional jumps before finally jumping to the final destination. The attacker will typically always detect the same set of taken jumps (i.e., all the unconditional jumps) and cannot distinguish the final jump from the decoy-jumps.

However, Van Bulck et al. [15] presented *SGX-Step*, a framework consisting of a Linux kernel driver and runtime library that manipulates the processor's Advanced Programmable Interrupt Controller (APIC) timer in order to interrupt an enclave after a single instruction i.e., to single-step the enclave's execution. They show that this makes the Zigzagger defense ineffective because the attacker can distinguish meaningful jumps from decoys.

3 SPECTRE MITIGATION TECHNIQUES

The recent Spectre [9] and SGXPectre [4] attacks are similar to branch-shadowing in that they abuse the BPU to exploit speculative execution. Whereas branch-shadowing aims to infer prior branching behavior, these attacks instead manipulate upcoming

branch prediction, e.g., cause speculative execution to touch otherwise inaccessible memory. Although not designed to do so, we suspected the new Spectre mitigation techniques could also affect the branch-shadowing attacks. However, our testing indicates that neither the recent firmware patches from Intel[6], nor the compiler-based Retpoline[7] affect the ability to perform branch-shadowing attacks against SGX.

In particular, we confirmed that Indirect Branch Restricted Speculation (IBRS) — designed to prevent unprivileged code from affecting speculation in privileged execution, e.g., within the enclave — has no effect on branch-shadowing. In our tests we saw no difference between an updated i7-7500U CPU and non-updated machines. We speculate that this is because IBRS is specifically designed to prevent low-privilege code from affecting high-privilege code. Whereas branch-shadowing relies on high-privileged code affecting in subsequent low-privilege code. The Retpoline defense replaces branch instructions with return instructions but our tests indicate that return statements affect the BTB, not only the dedicated Return Stack Buffer (RSB). SGXPectre further demonstrated that Spectre attacks can be performed against Retpoline.

4 THREAT MODEL AND REQUIREMENTS

We assume that the attacker has fine-grained control of enclave execution, i.e., can interrupt the enclave with instruction-level accuracy. The attacker can thus perform a branch-shadowing attack against every branch instruction. Specifically, the attacker can determine whether or not a branch instruction has been executed and taken (i.e., whether a conditional jump fell through or not). If the branching decisions depend on sensitive enclave data, the attacker can infer this data through the branch-shadowing attack.

This is a significantly stronger attacker capability than that assumed by previous work [10] because Van Bulck et al. [15] showed that single-step execution of SGX enclaves is both feasible to implement and sufficient to break existing defenses like Zigzagger [10]. We focus on branch-shadowing attacks and do not consider other side-channels, such as cache or page-fault attacks.

Given these attacker capabilities, we require a defence mechanism that prevents fine-grained branch-shadowing from revealing secret-dependent control flow. Specifically, in the instrumented code, we require that:

R.1 Any branch that can be directly observed through branch shadowing reveals no secret-dependent control flow information.

R.2 For any secret-dependent branches, the attacker's probability of success is bounded based on a security parameter k.

5 PROPOSED APPROACH

Our mitigation scheme uses compile-time obfuscation and run-time randomization to hide the control flow of an enclave application. While our proposed method is inspired by and uses a similar approach to Zigzagger, we assume a stronger attacker model. Specifically, our approach can defend against branch-shadowing even in the presence of an attacker with single-step capabilities.

[6] https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

[7] https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf

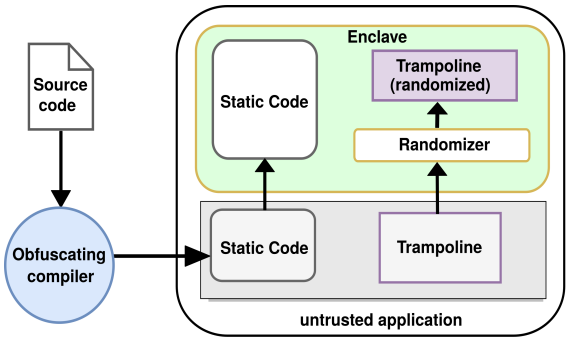


Figure 1: System design

Figure 1 illustrates the high-level view of our approach. The system consists of two main components: an obfuscating compiler and a run-time randomizer. The obfuscating compiler modifies the code by converting all branching instructions to indirect branches. The indirect branch targets are then explicitly set by the instrumentation depending on the converted branch type. We use conditional moves as replacements for conditional branches, allowing us to replicate the functionality of any conditional branch without involving the BPU. The observable control flow transitions, i.e., non trampoline branches, are further organized so that they are always unconditionally executed in the same order. The key insight of our approach is that, unlike Zigzagger, the trampolines are randomized inside the enclave at run-time by the randomizer. This prevents the attacker from reliably tracking their execution. Since only the trampolines are randomized, all other code remains in execute-only memory. Taken together, these two properties fulfill requirements **R.1** and **R.2**, as we show in our security evaluation in Section 7.

Listing 1 and Figure 2 show a single `if`-statement and corresponding Control Flow Graph. The corresponding obfuscated CFG is show in Figure 3. Figure 4 shows the same obfuscated code with the branch instructions converted. The static code is produced at compile time and its layout is assumed to be known to the attacker. The trampoline is similarly produced at compile time but is then randomized at run-time within the enclave. We assume that the attacker can observe and shadow the static code whereas the trampoline is unknown. Specifically, our approach works as follows:

Branch conversion: All branching instructions are converted to indirect unconditional branches. A register (`r15`) is reserved and populated with the original branch targets, which are stored in a jump-table that is updated during randomization. Conditional branches are converted to conditional moves (`cmov`) (e.g., `Block0` in Figure 4).

Jump blocks: Each block is followed by a *jump-block* that jumps to a trampoline indicated by `r15`. Execution flows that do not include a specific block still go through any intermediate jump-blocks to ensure that all indirect jumps outside the trampolines are executed. For instance, when taking the `if`-clause (`Block1`), the else-block (`Block2`) must not be executed but the corresponding jump-block (`B2J`) must be (e.g., the blue line in Figure 4). This ensures that an attacker always sees the same sequence of jumps (i.e., `B0J`, `B1J`, and `B2J`), regardless of actual executed code.

Listing 1: Example code before instrumentation

```
if (a !=0)
    /* Block1 */
else
    /* Block2 */
/* Block3 */
```

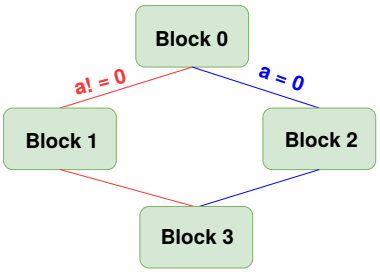


Figure 2: Original control flow graph

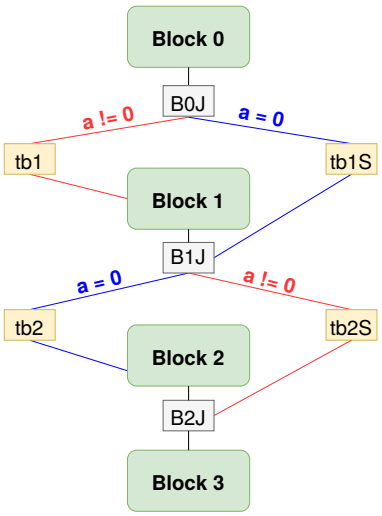


Figure 3: Modified control flow graph

Trampolines: The corresponding trampolines are created, corresponding to either the branching target or the fall-through block (i.e., the next block that will be executed when a conditional branch is not taken). In Figure 3, after execution of the if-block (`Block1`) the control flow is transferred to `tb2S` that will jump to the following jump-block `B2J` without executing the corresponding `Block2` itself.

Skip blocks: When skipping a block — e.g., the `else` block after taking the `if` block — we must nonetheless execute the corresponding jump-block to prevent its omission from leaking information. The jump-block target is prepared in the prior trampoline block by setting `r15`. For instance, after executing the if-block the corresponding trampoline (`tb2S`) not only jumps to the correct jump-block, but also sets the next target, `tb3`, into `r15`. To prevent timing attacks that measure the number of instructions between jump-blocks, the skipping trampolines (e.g., `tb1S` and `tb2S`) are populated with dummy-instructions to ensure that the timing between each jump-block is constant regardless of control flow. Although not shown in our example code, nested blocks are treated similarly to ensure that they execute all intermediary jumps.

Randomization: Trampolines are prepared during compilation, and are randomized at run-time inside the enclave. The randomization is implemented such that shadowing it does not reveal the randomization pattern. Randomizing the trampolines forces the attacker to shadow all possible locations in the enclave and thus, prevents shadowing the trampoline branches and reliably tracking the program's execution.

Re-randomization: Since an attacker could repeatedly call the same enclave functionality to gradually determine the randomization pattern, we can periodically re-randomize the trampolines. For example, the trampolines could be re-randomized on each enclave entry. As future work we envision to: a) provide code-annotation for limiting the obfuscation to only developer-determined sensitive parts, and b) randomize the trampoline code only when detecting multiple enclave entries (i.e., after a given number of potential shadowing attempts).

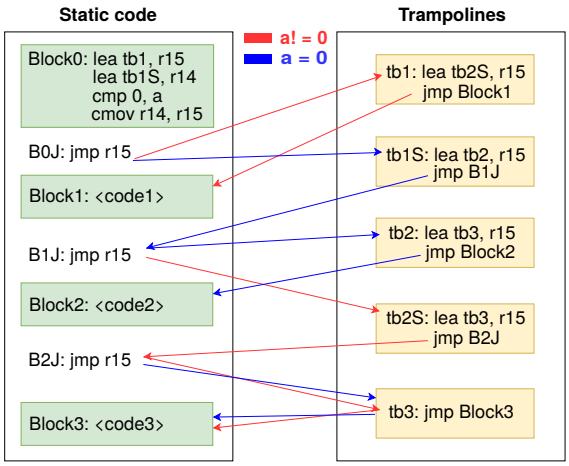


Figure 4: Modified code protected by our approach

6 IMPLEMENTATION DETAILS

We have implemented an open-source prototype of our approach, based on LLVM 6.0 and implemented in the X86 target backend. The instrumentation is applied by systematically traversing all functions and modifying their branching instructions, as explained in Section 5. Since the run-time randomization library cannot be randomized, it must be resistant to branch-shadowing attacks. While implemented, we have not yet integrated the randomizer to our instrumentation. For efficient and fine-grained randomization we do not preform in-place randomization, instead, we move trampoline entries between two trampoline areas. Listing 2 shows an overview of our randomization algorithm. Detailed description is available in our extended technical report [8].

We have also implemented an application for shadowing in-enclave execution in a controlled manner. Our setup is similar to [10] i.e., our application 1) retrieves branch instruction addresses

Listing 2: Randomization algorithm

```
for (entry = 0 : jump_table_entries) {
    location = rand() % trampoline_size;
    if (fits(entry, location))
        mark_reserved(location, entry);
    else
        l = (l+1) % trampoline_size;
    move_entry(entry, location);
}
```

and sets up a corresponding shadow-jump, 2) executes the victim enclave function and returns, 3) enables performance counters and executes the shadow-code, and 4) reads performance counters to infer in-enclave execution. Our setup is such that it could be integrated into the SGXStep-framework. We have replicated the shadowing techniques shown by [10] and performed shadowing on return statements.

7 EVALUATION

7.1 Security Analysis

As specified in Requirements (Section 4), we must prevent an attacker from inferring the secret-dependent control flow by **R.1**) ensuring that observable branches do not leak information, and **R.2**) preventing the attacker from probing other branches with a probability based on the security parameter k.

To hide any data-dependant branches (**R.1**), we replace all conditional branches with unconditional branches. We further setup the control flow so that each block in the static code section is executed in the same order and on each function call. One limitation is that we do not conceal the number of loop executions, because this is typically unknown compile time. In some cases this could be avoided by unrolling loops.

The remaining branching instructions are exclusively in the trampolines, for which the locations are randomized to defend against shadowing (**R.2**). Without knowing the exact trampoline layout, the attacker is forced to guess or exhaustively probe all possible locations. The probability of attack success (P_{attack}) is given by $P_{attack} = \frac{G}{k}$, where G is the number of guesses and k the number of possible trampoline locations.

The upper limit for G is the number of BTB entries, but in practice this is lowered by any intermediate code (e.g., system calls and attack setup) that pollutes the BTB. The security parameter k determines the trampoline randomization space. Because X86 allows unaligned execution, a single 4KB range gives us up to 4091 potential trampoline locations (with a trampoline size starting at 5 bytes). With a randomization area of 8KB and 4096 BTB entries, the success probability of shadowing a single branch has an upper bound of 0.5. The probability of following the full control flow drops exponentially as the number of targeted branches increase.

7.2 Performance Evaluation

We evaluated the overhead of our system in terms of CPU-utilization, memory use, and code size. All software was compiled using the SGX SDK version 2.0 and run on an SGX-enabled Intel Skylake Core i5-6500 CPU clocked at 3.20 GHz, with 7,6 GiB of RAM, running Ubuntu 16.04 with a 64-bit Linux 4.4.0-96-generic kernel.

Table 1: Computational performance (iterations/second) before and after instrumentation, excluding randomization and dummy instructions.

Benchmark	Before (std. dev.)	After (std. dev.)	Performance loss
Numeric sort	828.8 (0.79)	578.8 (0.21)	30%
String sort	86.59 (0.09)	67.72 (0.21)	21%
Bitfield	1.839e8 (1.34e5)	1.370e8 (3.27e5)	25%
Fp emulation	87.70 (0.11)	42.73 (0.02)	51%
Fourier	1.789e5 (1.19e2)	1.500e5 (1.50e2)	16%
Assignment	21.64 (0.03)	7.769 (0.01)	64%
Idea	2667 (1.26)	2665 (1.84)	0.1%
Huffman	2354 (4.07)	860.5 (0.71)	63%
Neural net	35.16 (0.03)	25.57 (0.22)	27%
Lu decomp	973.1 (1.45)	785.0 (1.41)	19%
Geometric mean			17.17%

We used SGX-Nbench[8] which is adapted from Nbench-byte-2.2.3, to measure the CPU and memory overhead of 10 different benchmarks executed within an enclave. All benchmarks were conducted with full instrumentation, but do not include randomization or dummy-instructions. Although the randomization would introduce additional overhead, it need not be constantly repeated. Instead it can be performed once on enclave creation and then later after a specified number of enclave re-entries.

CPU overhead: Table 1 shows the computational performance of various benchmarks in the enclave before and after obfuscation. The decrease in performance (i.e., the number of iterations per second) results from the addition of trampoline jumps and the need to exhaustively execute all jump-blocks. However, since we have obfuscated the entire program, these results represent the worst case scenarios. In real deployments, only the parts of the code that depend on secret data would be obfuscated. The performance penalty depends on how complicated the function is in terms of size and number of branches. The Assignment benchmark, for instance, has functions with many nested conditional branches, all of which require corresponding jump-blocks to be added and executed.

Memory overhead: As expected, our instrumentation does not increase heap or stack usage of the enclave.

Code size: To measure the increase in code size, we compared the size of the enclave object files before and after instrumentation. The size of the SGX-Nbench object files increased from 329.1 kB to 370.1 kB after instrumentation. Similarly to performance overhead, code size overhead will also decrease when instrumenting only the secret-dependent sections of the code.

8 RELATED WORK

There is a growing body of research on side channel attacks targeting Intel SGX and corresponding countermeasures. In addition to the branch-shadowing attacks [5, 10], there are other side channel attacks targeting SGX enclaves [2, 7, 14, 16].

[8] https://github.com/utds3lab/sgx-nbench

Several approaches have been presented to thwart controlled-channel (page-fault) attacks. *SGX-Shield* [11] randomizes the memory layout, similar to Address Space Layout Randomization (ASLR), to prevent control flow hijacking and hide the enclave memory layout. This approach impedes run-time attacks that exploit memory errors or attacks that rely on a known memory layout (e.g., controlled-channel attacks). SGX-Shield uses on-load randomization, allowing repeated branch-shadowing attacks to gradually reveal the randomization pattern. Our approach solves this through run-time re-randomization. We further minimize the additional attack-surface by limiting the randomization to the trampolines.

Shinde et al. [13] propose an approach that masks page-fault patterns by making the program's memory access pattern deterministic. More precisely, they alter the program such that it accesses all its data and code pages in the same sequence, regardless of the input. This makes the enclave application demonstrate the same page-fault pattern for any secret input variables. *T-SGX* [12] leverages Intel Transactional Synchronization Extensions (TSX) to suppress encountered page-faults without invoking the underlying OS. Although T-SGX does not mitigate branch-shadowing attacks [10], it could be combined with our approach to address both branch-shadowing and page-fault attacks.

DR.SGX [1] is presented to defend against cache side-channel attacks. It permutes data locations, and continuously re-randomizes enclave data in order to hamper correlation of memory accesses. This approach prevents leakages resulting from secret-dependant data accesses. Similarly, Chandra et. al [3] inject dummy data instances into the user-supplied data instances in order to add noise to memory access traces. They randomize/shuffle the dummy data with the user data to reduce the chance of extracting sensitive information from side-channels. Both approaches are similar to ours in that they employ randomization, but they are not designed to defend against branch shadowing attacks since they randomize data memory locations rather than control flow.

CCFIR (Compact Control Flow Integrity and Randomization) [17] is a new method proposed to impede control-flow hijacking attacks (e.g., return-into-libc and ROP). CCFIR controls the indirect control transfers and limits the possible jump location to a whitelist in a Springboard. Randomizing the order of the stubs in the Springboard adds an extra layer of protection and frustrates guessing of the function pointers and return addresses. However, CCFIR has not been designed for use in SGX enclaves.

Obfuscation techniques were previously used to thwart leakages via side-channel attacks. Oblivious RAM (ORAM) [6] conceals the program's memory access pattern by shuffling and re-encrypting the accessed data. However, the state should be stored/updated at client-side, which makes it difficult to use for protecting cache since it is challenging to store the internal state of ORAM securely without hardware support, given the small size of cache lines. Moreover, this approach incurs significant performance overhead.

None of the above countermeasures focus on mitigating branch-shadowing attacks, and additionally, Lee et. al [10] have demonstrated that their branch-shadowing attack is capable of breaking the security constructs of SGX-Shield, T-SGX, and ORAM.

9 CONCLUSION AND FUTURE WORK

We propose a software-based mitigation scheme to defend against branch-shadowing attacks, even in the presence of attackers with the ability to single-step through SGX enclaves. Our approach combines compile-time control flow obfuscation with run-time code randomization to prevent the enclave program from leaking secret-dependant control flow. We evaluated our approach using ten benchmarks from SGX-Nbench. Although we considered the worst-case scenario (whole program instrumentation), our results show that, on average, our approach results in less than 18% performance loss and less than 1.2 times code size increase.

As future work, we will integrate the randomizing component and optimize our obfuscating compiler to reduce overhead. In addition, we plan to integrate our approach with other defences, in order to mitigate a broader range of side-channel attacks.

ACKNOWLEDGMENTS

This work was supported in part by the Intel Collaborative Research Institute for Collaborative Autonomous and Resilient Systems (ICRI-CARS) at Aalto University.

REFERENCES

[1] F. Brasser et al. 2017. DR.SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. (2017). http://arxiv.org/abs/1709.09917
[2] F. Brasser et al. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies*. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser
[3] S. Chandra et al. 2017. Securing Data Analytics on SGX with Randomization. In *22nd European Symposium on Research in Computer Security*. https://doi.org/10.1007/978-3-319-66402-6_21
[4] G. Chen et al. 2018. SGXPECTRE Attacks: Leaking Enclave Secrets via Speculative Execution. (2018). https://arxiv.org/abs/1802.09085
[5] D. Evtyushkin et al. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems*. https://doi.org/10.1145/3173162.3173204
[6] O. Goldreich and R. Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431 – 473. https://doi.org/10.1145/233551.233553
[7] J. Götzfried et al. 2017. Cache Attacks on Intel SGX. In *10th European Workshop on Systems Security*. https://doi.org/10.1145/3065913.3065915
[8] S. Hosseinzadeh et al. 2018. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. (2018). https://arxiv.org/abs/1808.06478
[9] P. Kocher et al. 2018. Spectre Attacks: Exploiting Speculative Execution. (2018). https://spectreattack.com/spectre.pdf
[10] S. Lee et al. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho
[11] J. Seo et al. 2017. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2017.23037
[12] M.-W. Shih et al. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Network and Distributed System Security Symposium*. https://doi.org/10.14722/ndss.2017.23193
[13] S. Shinde et al. 2015. Preventing Your Faults From Telling Your Secrets: Defenses Against Pigeonhole Attacks. (2015). http://arxiv.org/abs/1506.04832
[14] J. Van Bulck et al. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *26th USENIX Security Symposium*. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/van-bulck
[15] J. Van Bulck, F. Piessens, and R. Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *2nd Workshop on System Software for Trusted Execution*. https://doi.org/10.1145/3152701.3152706
[16] Y. Xu, W. Cui, and M. Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2015.45
[17] C. Zhang et al. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2013.44

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services
178. **Jari Björne**, Biomedical Event Extraction with Machine Learning
179. **Katri Haverinen**, Natural Language Processing Resources for Finnish: Corpus Development in the General and Clinical Domains
180. **Ville Salo**, Subshifts with Simple Cellular Automata
181. **Johan Ersfolk**, Scheduling Dynamic Dataflow Graphs
182. **Hongyan Liu**, On Advancing Business Intelligence in the Electricity Retail Market
183. **Adnan Ashraf**, Cost-Efficient Virtual Machine Management: Provisioning, Admission Control, and Consolidation
184. **Muhammad Nazrul Islam**, Design and Evaluation of Web Interface Signs to Improve Web Usability: A Semiotic Framework
185. **Johannes Tuikkala**, Algorithmic Techniques in Gene Expression Processing: From Imputation to Visualization
186. **Natalia Díaz Rodríguez**, Semantic and Fuzzy Modelling for Human Behaviour Recognition in Smart Spaces. A Case Study on Ambient Assisted Living
187. **Mikko Pänkäälä**, Potential and Challenges of Analog Reconfigurable Computation in Modern and Future CMOS
188. **Sami Hyrynsalmi**, Letters from the War of Ecosystems – An Analysis of Independent Software Vendors in Mobile Application Marketplaces
189. **Seppo Pulkkinen**, Efficient Optimization Algorithms for Nonlinear Data Analysis
190. **Sami Pyöttiälä**, Optimization and Measuring Techniques for Collect-and-Place Machines in Printed Circuit Board Industry
191. **Syed Mohammad Asad Hassan Jafri**, Virtual Runtime Application Partitions for Resource Management in Massively Parallel Architectures
192. **Toni Ernvall**, On Distributed Storage Codes
193. **Yuliya Prokhorova**, Rigorous Development of Safety-Critical Systems
194. **Olli Lahdenoja**, Local Binary Patterns in Focal-Plane Processing – Analysis and Applications
195. **Annika H. Holmbom**, Visual Analytics for Behavioral and Niche Market Segmentation
196. **Sergey Ostroumov**, Agent-Based Management System for Many-Core Platforms: Rigorous Design and Efficient Implementation
197. **Espen Suenson**, How Computer Programmers Work – Understanding Software Development in Practise
198. **Tuomas Poikela**, Readout Architectures for Hybrid Pixel Detector Readout Chips
199. **Bogdan Iancu**, Quantitative Refinement of Reaction-Based Biomodels
200. **Ilkka Törmä**, Structural and Computational Existence Results for Multidimensional Subshifts
201. **Sebastian Okser**, Scalable Feature Selection Applications for Genome-Wide Association Studies of Complex Diseases
202. **Fredrik Abbors**, Model-Based Testing of Software Systems: Functionality and Performance
203. **Inna Pereverzeva**, Formal Development of Resilient Distributed Systems
204. **Mikhail Barash**, Defining Contexts in Context-Free Grammars
205. **Sepinoud Azimi**, Computational Models for and from Biology: Simple Gene Assembly and Reaction Systems
206. **Petter Sandvik**, Formal Modelling for Digital Media Distribution

207. **Jongyun Moon**, Hydrogen Sensor Application of Anodic Titanium Oxide Nanostructures
208. **Simon Holmbacka**, Energy Aware Software for Many-Core Systems
209. **Charalampos Zinoviadis**, Hierarchy and Expansiveness in Two-Dimensional Subshifts of Finite Type
210. **Mika Murtojärvi**, Efficient Algorithms for Coastal Geographic Problems
211. **Sami Mäkelä**, Cohesion Metrics for Improving Software Quality
212. **Eyal Eshet**, Examining Human-Centered Design Practice in the Mobile Apps Era
213. **Jetro Vesti**, Rich Words and Balanced Words
214. **Jarkko Peltomäki**, Privileged Words and Sturmian Words
215. **Fahimeh Farahnakian**, Energy and Performance Management of Virtual Machines: Provisioning, Placement and Consolidation
216. **Diana-Elena Gratie**, Refinement of Biomodels Using Petri Nets
217. **Harri Merisaari**, Algorithmic Analysis Techniques for Molecular Imaging
218. **Stefan Grönroos**, Efficient and Low-Cost Software Defined Radio on Commodity Hardware
219. **Noora Nieminen**, Garbling Schemes and Applications
220. **Ville Taajamaa**, O-CDIO: Engineering Education Framework with Embedded Design Thinking Methods
221. **Johannes Holvitie**, Technical Debt in Software Development – Examining Premises and Overcoming Implementation for Efficient Management
222. **Tewodros Deneke**, Proactive Management of Video Transcoding Services
223. **Kashif Javed**, Model-Driven Development and Verification of Fault Tolerant Systems
224. **Pekka Naula**, Sparse Predictive Modeling – A Cost-Effective Perspective
225. **Antti Hakkala**, On Security and Privacy for Networked Information Society – Observations and Solutions for Security Engineering and Trust Building in Advanced Societal Processes
226. **Anne-Maarit Majanoja**, Selective Outsourcing in Global IT Services – Operational Level Challenges and Opportunities
227. **Samuel Rönnqvist**, Knowledge-Lean Text Mining
228. **Mohammad-Hashem Hahgbayan**, Energy-Efficient and Reliable Computing in Dark Silicon Era
229. **Charmi Panchal**, Qualitative Methods for Modeling Biochemical Systems and Datasets: The Logicome and the Reaction Systems Approaches
230. **Erkki Kaila**, Utilizing Educational Technology in Computer Science and Programming Courses: Theory and Practice
231. **Fredrik Robertsén**, The Lattice Boltzmann Method, a Petaflop and Beyond
232. **Jonne Pohjankukka**, Machine Learning Approaches for Natural Resource Data
233. **Paavo Nevalainen**, Geometric Data Understanding: Deriving Case-Specific Features
234. **Michal Szabados**, An Algebraic Approach to Nivat's Conjecture
235. **Tuan Nguyen Gia**, Design for Energy-Efficient and Reliable Fog-Assisted Healthcare IoT Systems
236. **Anil Kanduri**, Adaptive Knobs for Resource Efficient Computing
237. **Veronika Suni**, Computational Methods and Tools for Protein Phosphorylation Analysis
238. **Behailu Negash**, Interoperating Networked Embedded Systems to Compose the Web of Things
239. **Kalle Rindell**, Development of Secure Software: Rationale, Standards and Practices
240. **Jurka Rahikkala**, On Top Management Support for Software Cost Estimation
241. **Markus A. Whiteland**, On the k-Abelian Equivalence Relation of Finite Words
242. **Mojgan Kamali**, Formal Analysis of Network Routing Protocols
243. **Jesús Carabaño Bravo**, A Compiler Approach to Map Algebra for Raster Spatial Modeling
244. **Amin Majd**, Distributed and Lightweight Meta-heuristic Optimization Method for Complex Problems
245. **Ali Farooq**, In Quest of Information Security in Higher Education Institutions: Security Awareness, Concerns, and Behaviour of Students
246. **Juho Heimonen**, Knowledge Representation and Text Mining in Biomedical, Healthcare, and Political Domains

247. **Sanaz Rahimi Moosavi**, Towards End-to-End Security in Internet of Things based Healthcare
248. **Mingzhe Jiang**, Automatic Pain Assessment by Learning from Multiple Biopotentials
249. **Johan Kopra**, Cellular Automata with Complicated Dynamics
250. **Iman Azimi**, Personalized Data Analytics for Internet-of-Things-based Health Monitoring
251. **Jaakko Helminen**, Systems Action Design Research: Delineation of an Application to Develop Hybrid Local Climate Services
252. **Aung Pyae**, The Use of Digital Games to Enhance the Physical Exercise Activity of the Elderly: A Case of Finland
253. **Woubishet Zewdu Taffese**, Data-Driven Method for Enhanced Corrosion Assessment of Reinforced Concrete Structures
254. **Etienne Moutot**, Around the Domino Problem – Combinatorial Structures and Algebraic Tools
255. **Joonatan Jalonen**, On Some One-Sided Dynamics of Cellular Automata
256. **Outi Montonen**, On Multiobjective Optimization from the Nonsmooth Perspective
257. **Tuomo Lehtilä**, On Location, Domination and Information Retrieval
258. **Shohreh Hosseinzadeh**, Security and Trust in Cloud Computing and IoT through Applying Obfuscation, Diversification, and Trusted Computing Technologies

Turku Centre *for* Computer Science

http://www. tucs.fi

tucs@abo.fi

University of Turku
Faculty of Science and Engineering
- Department of Future Technologies
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Faculty of Science and Engineering
- Computer Engineering
- Computer Science

Faculty of Social Sciences, Business and Economics
- Information Systems

ISBN 978-952-12-3992-2
ISSN 1239-1883

Shohreh Hosseinzadeh

Security and Trust in Cloud Computing and IoT

Shohreh Hosseinzadeh

Security and Trust in Cloud Computing and IoT

Shohreh Hosseinzadeh

Security and Trust in Cloud Computing and IoT