Welcome to Cambridge

Alastair Beresford


UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology
“contribute to society through the pursuit
of education, learning and research at the
highest international levels of excellence”

Computing: 88 years of research and 72 years of teaching

1937 Dept. founded as Mathematical Laboratory

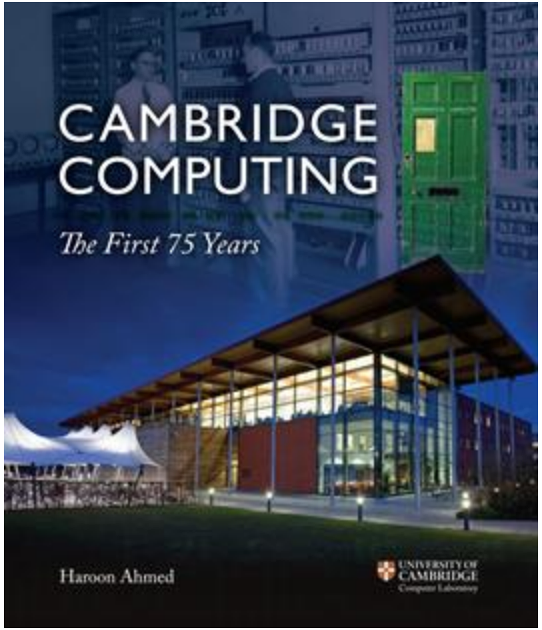
1949 Program-stored computer, EDSAC

1951 World’s first PhD (Wheeler)

1953 World’s first computer science course (Diploma)

1971 Computer Science Tripos launched

1985 MPhil, initially with Engineering



https://www.cl.cam.ac.uk/downloads/books/CambridgeComputing_Ahmed.pdf

LEO: an early example of industrial collaboration

LEO I
BBC Archives

Photo credit: J Lyons & Co

The Centre for Computing History: https://www.youtube.com/watch?v=Rzu68nRVwtE

Hall of Fame companies

Staff and alumni founded 340+ companies:

- £7.6bn+ annual revenue
- £138bn+ combined valuation
- 28k+ employees

With thanks for the analysis:

Beauhurst

Changing the world takes many forms

- Significant research breakthroughs transition to established industrial partner
- Radical rethinking of a commercial product
- Addressing a gap in the market with a start-up
- Open sourcing software to unlock value
- Charitable missions
- …

EDSAC (1940s)

Titan (1960s)

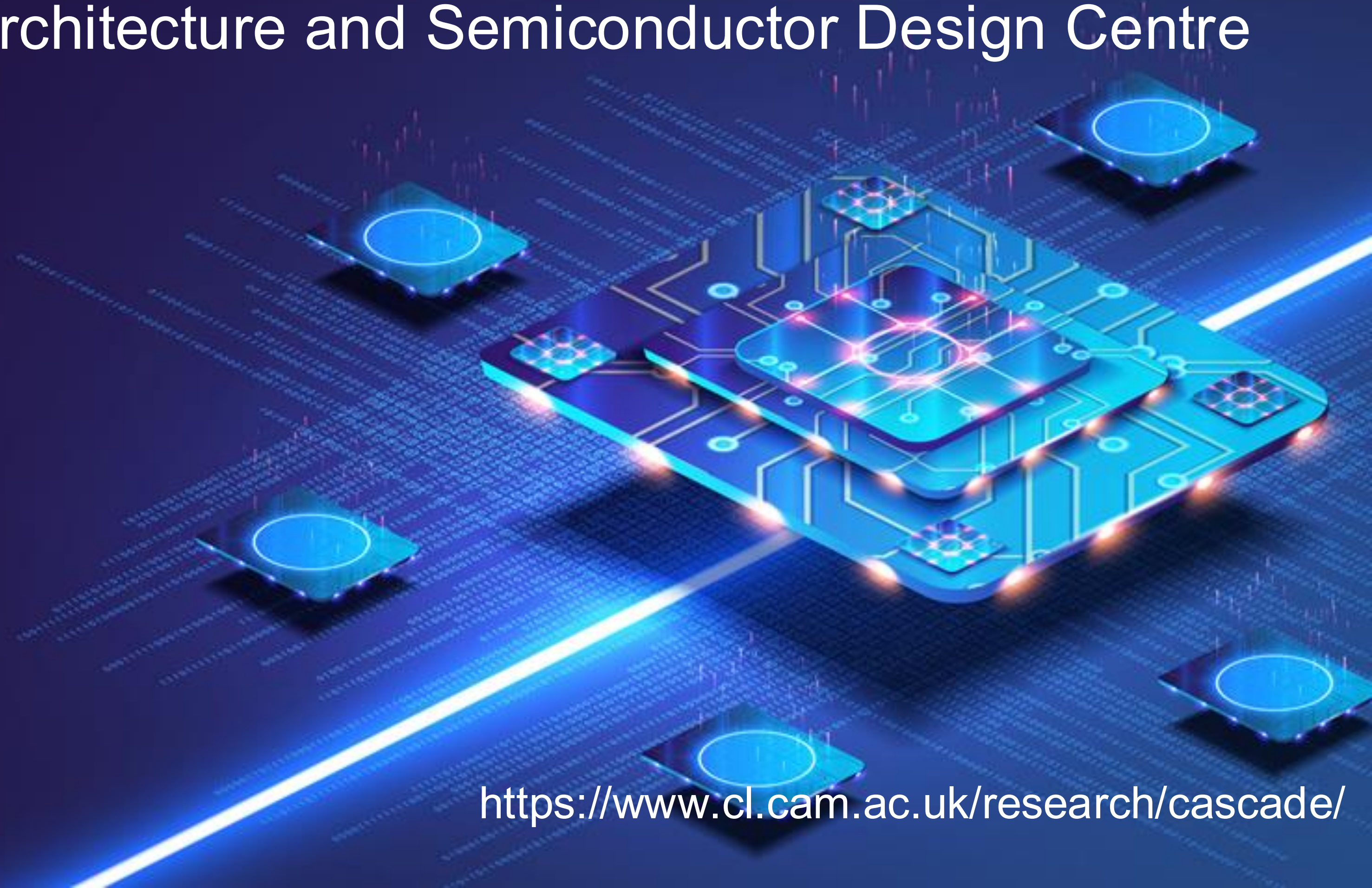
BBC Micro (1980s)

Xen Project

Xen Source (2000s)

Raspberry

Computer Architecture and Semiconductor Design Centre
CASCADE



https://www.cl.cam.ac.uk/research/cascade/

Does software behave more like milk or wine?

- Studied OpenBSD over 7.5 years
- 61% code was *foundational*
- Decrease in foundational vulns over time
- Median vuln lifetime of 2.6 years

Milk or Wine: Does Software Security Improve with Age? [*,†]

Andy Ozment
MIT Lincoln Laboratory[‡]

Stuart E. Schechter
MIT Lincoln Laboratory

Abstract

We examine the code base of the OpenBSD operating system to determine whether its security is increasing over time. We measure the rate at which new code has been introduced and the rate at which vulnerabilities have been reported over the last 7.5 years and fifteen versions.

We learn that 61% of the lines of code in today's OpenBSD are *foundational*: they were introduced prior to the release of the initial version we studied and have not been altered since. We also learn that 62% of reported vulnerabilities were present when the study began and can also be considered to be foundational.

We find strong statistical evidence of a decrease in the rate at which foundational vulnerabilities are being reported. However, this decrease is anything but brisk: foundational vulnerabilities have a median lifetime of *at least* 2.6 years.

Finally, we examined the density of vulnerabilities in the code that was altered/introduced in each version. The densities ranged from 0 to 0.033 vulnerabilities reported per thousand lines of code. These densities will increase as more vulnerabilities are reported.

*This work is sponsored by the I3P under Air Force Contract FA8721-05-0002. Opinions, interpretations, conclusions and recommendations are those of the author(s) and are not necessarily endorsed by the United States Government.

†This work was produced under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College, and supported under Award number 2003-TK-TX-0003 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the authors and do not necessarily represent the official position of the U.S. Department of Homeland Security, the Science and Technology Directorate, the I3P, or Dartmouth College.

‡Currently at the University of Cambridge

1 Introduction

Many in the security research community have criticized both the insecurity of software products and developers' perceived inattention to security. However, we have lacked quantitative evidence that such attention can improve a product's security over time. Seeking such evidence, we asked whether efforts by the OpenBSD development team to secure their product have decreased the rate at which vulnerabilities are reported.

In particular, we are interested in responding to the work of Eric Rescorla [11]. He used data from ICAT[1] to argue that the rate at which vulnerabilities are reported has not decreased with time; however, limitations in the data he used prompted us to investigate this area further.

We chose OpenBSD version 2.3 as our *foundation version*, and we collected 7.5 years of data on the vulnerabilities reported in OpenBSD since that version's release. In particular, we focused our analysis on *foundational vulnerabilities*: those introduced prior to the release of the foundation version. We also analyzed the evolution of the code base. We were driven by the goal of answering the following questions:

1. How much does legacy code influence security today?
2. Do larger code changes have more vulnerabilities?
3. Do today's coders introduce fewer vulnerabilities per line of code?
4. What is the median lifetime of a vulnerability?

Most importantly:

5. Has there been a decline in the rate at which foundational vulnerabilities in OpenBSD are reported?

In the upcoming section, we discuss the limitations of vulnerability reporting data; these limitations may result in our analysis underestimating increases in the security

https://www.usenix.org/legacy/events/sec06/tech/full_papers/ozment/ozment.pdf

Security is a game of whack-a-mole

Empirical analysis:
we are not good at patching

- ⅓ of issues introduced >3 years previously
- Patching takes months
- 5% of patches had negative impacts
- 7% of patches failed to remedy issue

A Large-Scale Empirical Study of Security Patches

Frank Li Vern Paxson
{frankli, vern}@cs.berkeley.edu
University of California, Berkeley and International Computer Science Institute

ABSTRACT

Given how the "patching treadmill" plays a central role for enabling sites to counter emergent security concerns, it behooves the security community to understand the patch development process and characteristics of the resulting fixes. Illumination of the nature of security patch development can inform us of shortcomings in existing remediation processes and provide insights for improving current practices. In this work we conduct a large-scale empirical study of security patches, investigating more than 4,000 bug fixes for over 3,000 vulnerabilities that affected a diverse set of 682 open-source software projects. For our analysis we draw upon the National Vulnerability Database, information scraped from relevant external references, affected software repositories, and their associated security fixes. Leveraging this diverse set of information, we conduct an analysis of various aspects of the patch development life cycle, including investigation into the duration of impact a vulnerability has on a code base, the timeliness of patch development, and the degree to which developers produce safe and reliable fixes. We then characterize the nature of security fixes in comparison to other non-security bug fixes, exploring the complexity of different types of patches and their impact on code bases.

Among our findings we identify that: security patches have a lower footprint in code bases than non-security bug patches; a third of all security issues were introduced more than 3 years prior to remediation; attackers who monitor open-source repositories can often get a jump of weeks to months on targeting not-yet-patched systems prior to any public disclosure and patch distribution; nearly 5% of security fixes negatively impacted the associated software; and 7% failed to completely remedy the security hole they targeted.

1 INTRODUCTION

Miscreants seeking to exploit computer systems incessantly discover and weaponize new security vulnerabilities. As malicious attacks become increasingly advanced, system administrators continue to rely on many of the same processes as practiced for decades to update their software against the latest threats. Given the central role that the "patching treadmill" plays in countering emergent security concerns, it behooves the security community to understand the patch development process and the characteristics of the resulting fixes. Illuminating the nature of security patch development can inform us of shortcomings in existing remediation processes and provide insights for improving current practices.

Seeking such understanding has motivated several studies exploring various aspects of vulnerability and patching life cycles. Some have analyzed public documentation about vulnerabilities, such as security advisories, to shed light on the vulnerability disclosure process [14, 32]. These studies, however, did not include analyses of the corresponding code bases and the patch development process itself. Others have tracked the development of specific projects to better understand patching dynamics [18, 28, 41]. While providing insights on the responsiveness of particular projects to security issues, these investigations have been limited to a smaller scale across a few (often one) projects.

Beyond the patch development life cycle, the characteristics of security fixes themselves are of particular interest, given their importance in securing software and the time sensitivity of their development. The software engineering community has studied bug fixes in general [29, 33, 34, 42]. However, there has been little investigation into how fixes vary across different classes of issues. For example, one might expect that patches for performance issues qualitatively differ from those remediating vulnerabilities. Indeed, Zama et al.'s case study on Mozilla Firefox bugs revealed that developers address different classes of bugs differently [41].

In this work, we conduct a large-scale empirical study of security patches, investigating 4,000+ bug fixes for 3,000+ vulnerabilities that affected a diverse set of 682 open-source software projects. We build our analysis on a dataset that merges vulnerability entries from the National Vulnerability Database [37], information scraped from relevant external references, affected software repositories, and their associated security fixes. Tying together these disparate data sources allows us to perform a deep analysis of the patch development life cycle, including investigation of the code base life span of vulnerabilities, the timeliness of security fixes, and the degree to which developers can produce safe and reliable security patches. We also extensively characterize the security fixes themselves in comparison to other non-security bug patches, exploring the complexity of different types of patches and their impact on code bases.

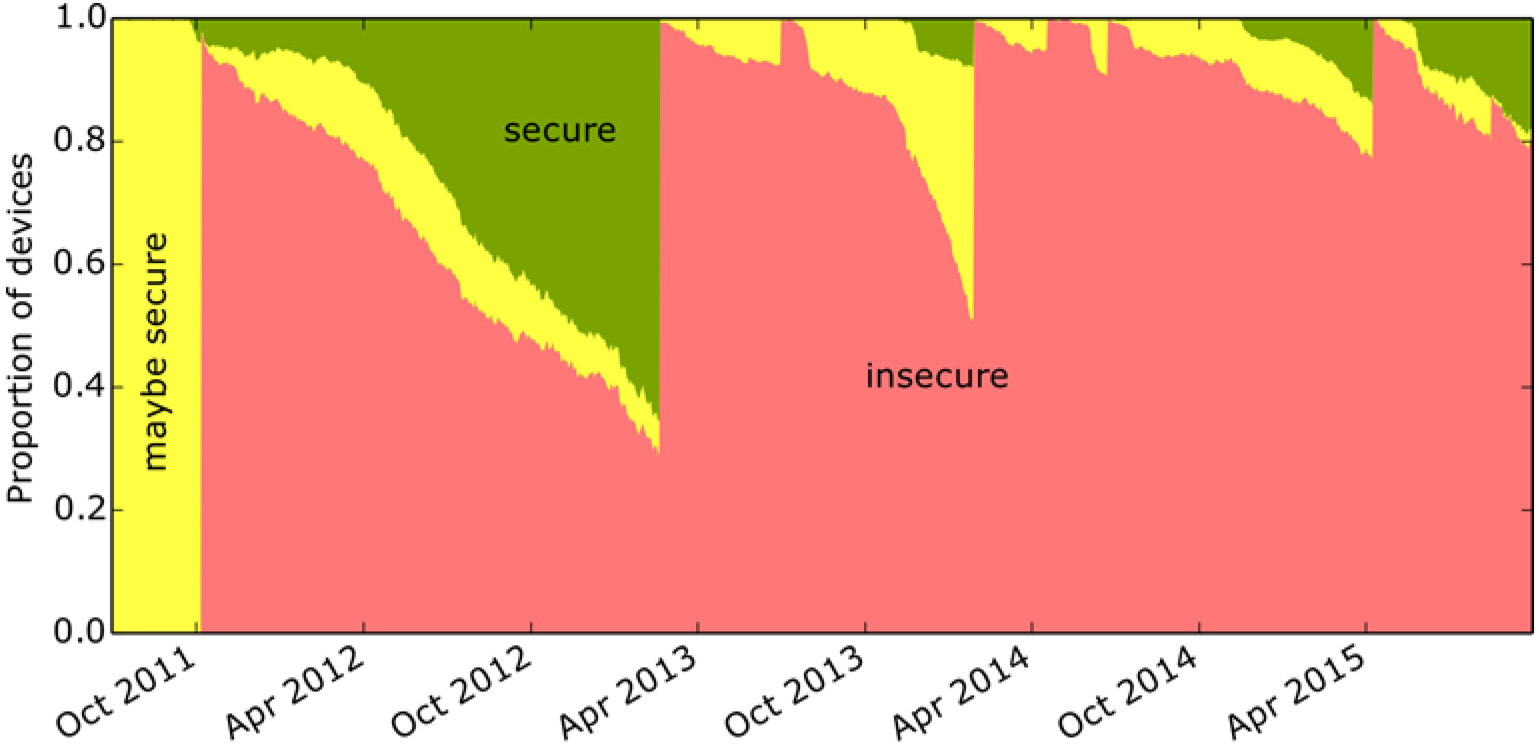
Among our findings we identify that: security patches have less impact on code bases and result in more localized changes than non-security bug patches; security issues reside in code bases for years, with a third introduced more than 3 years prior to remediation; security fixes are poorly timed with public disclosures, allowing attackers who monitor open-source repositories to get a jump of weeks to months on targeting not-yet-patched systems prior to any public disclosure and patch distribution; nearly 5% of security

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10...$15.00
https://doi.org/10.1145/3133956.3134072

https://dl.acm.org/doi/pdf/10.1145/3133956.3134072

Android handsets 2011-2016: 85% insecure



https://dl.acm.org/doi/abs/10.1145/2808117.2808118

Matt Welsh: in a few years nobody will write code any more

viewpoints

DOI:10.1145/3570220

Matt Welsh

Viewpoint

The End of Programming

The end of classical computer science is coming, and most of us are dinosaurs waiting for the meteor to hit.

I came of age in the 1980s, programming personal computers such as the Commodore VIC-20 and Apple][e at home. Going on to study computer science (CS) in college and ultimately getting a Ph.D. at Berkeley, the bulk of my professional training was rooted in what I will call "classical" CS: programming, algorithms, data structures, systems, programming languages. In Classical Computer Science, the ultimate goal is to reduce an idea to a program written by a human—source code in a language like Java or C++ or Python. Every idea in Classical CS—no matter how complex or sophisticated, from a database join algorithm to the mind-bogglingly obtuse Paxos consensus protocol—can be expressed as a human-readable, human-comprehendible program.

When I was in college in the early 1990s, we were still in the depths of the AI Winter, and AI as a field was likewise dominated by classical algorithms. My first research job at Cornell University was working with Dan Huttenlocher, a leader in the field of computer vision (and now Dean of the MIT Schwarzman College of Computing). In Huttenlocher's Ph.D.-level computer vision course in 1995 or so, we never once discussed anything resembling deep learning or neural networks—it was all classical algorithms like Canny edge detection, optical flow, and Hausdorff distances. Deep learning was in its infancy, not yet considered mainstream AI, let alone mainstream CS.

Of course, this was 30 years ago, and a lot has changed since then, but one thing that has not really changed is that CS is taught as a discipline with data structures, algorithms, and programming at its core. I am going to be amazed if in 30 years, or even 10 years, we are still approaching CS in this way. Indeed, I think CS as a field is in for a pretty major upheaval few of us are really prepared for.

Programming will be obsolete. I believe the conventional idea of "writing a program" is headed for extinction, and indeed, for all but very specialized applications, most software, as we know it, will be replaced by AI systems that are *trained* rather than *programmed*. In situations where one needs a "simple" program (after all, not everything should require a model of hundreds of billions of parameters running on a cluster of GPUs), those programs will, themselves, be generated by an AI rather than coded by hand.

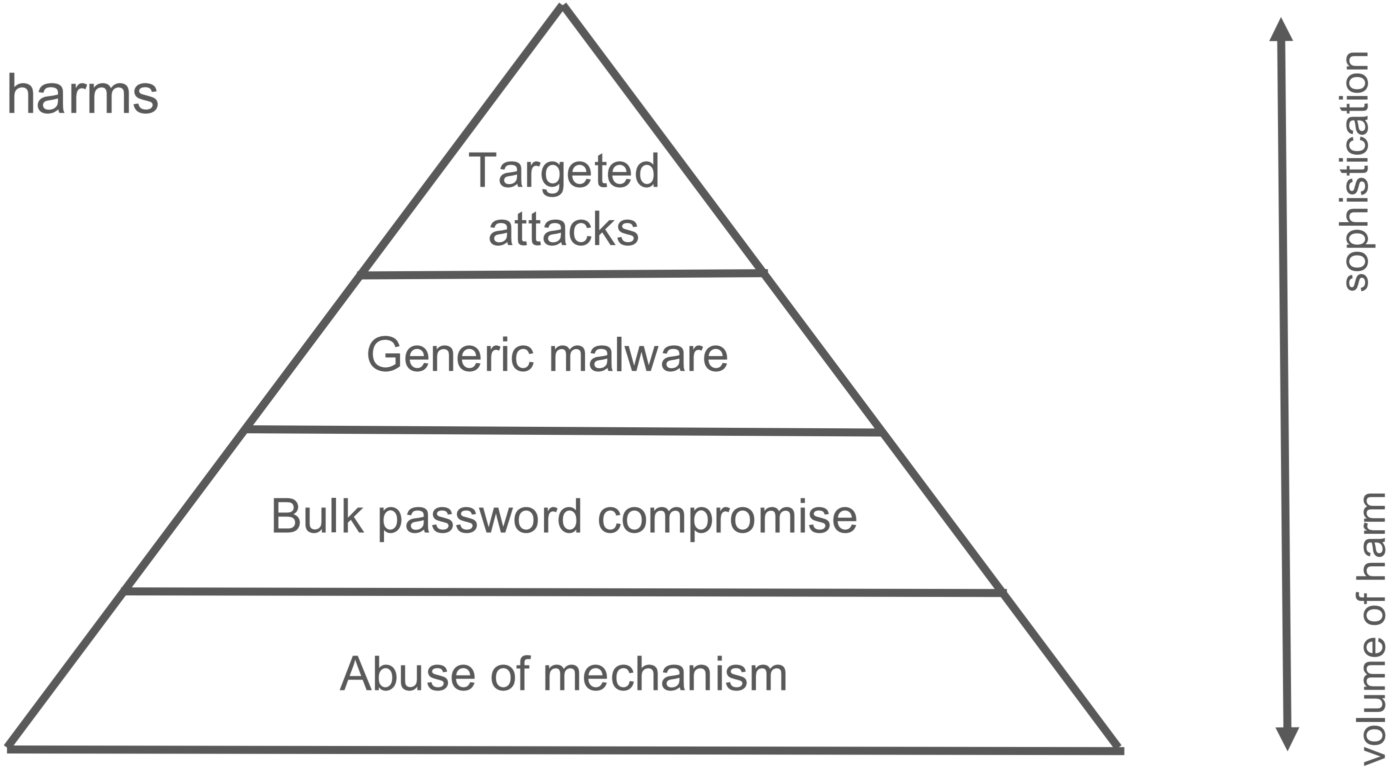
I do not think this idea is crazy. No doubt the earliest pioneers of computer science, emerging from the (relatively) primitive cave of electrical engineering, stridently believed that all future computer scientists would need to command a deep understanding of semiconductors, binary arithmetic, and microprocessor design to understand software. Fast-forward to today, and I am willing to bet good money that 99% of people who are writing software have almost no clue how a CPU actually works, let alone the physics underlying transistor design. By extension, I believe the computer scientists of the future will be so far removed from the classic definitions of "software" that they would be hard-pressed to reverse a linked list or implement Quicksort. (I am not sure I remember how to implement Quicksort myself.)

AI coding assistants such as CoPilot are only scratching the surface of what I am describing. It seems totally obvious to me that *of course* all programs in the

https://dl.acm.org/doi/pdf/10.1145/3570220

CHERI: necessary but not sufficient for secure systems

Hierarchy of harms




UNIVERSITY OF
CAMBRIDGE
Department of Computer
Science and Technology
“contribute to society through the pursuit
of education, learning and research at the
highest international levels of excellence”