Horizon 2020 Program (2014-2020)

Cybersecurity, Trustworthy ICT Research & Innovation Actions
Security-by-design for end-to-end security
ICT 32-2014

Secure Hardware-Software Architectures for
Robust Computing Systems [†]

Deliverable D4.1: Requirements of the SHARCS Runtime System, Software Tools and Reporting

Abstract: This deliverable describes the system architectures of SHARCS applications and their **software security requirements** aligned with the application requirements, as analyzed in WP2. It further reports on the progress of software hardening techniques for source code and binaries.

Contractual Date of Delivery	Month 12
Actual Date of Delivery	Month 12
Deliverable Dissemination Level	Public
Editor	Elias Athanasopoulos & Cristiano Giuffrida
Contributors	All *SHARCS* partners
Quality Assurance	John Thompson, Michail Flouris, and Thomas Kamm

[†] The research leading to these results has received funding from the European Union Horizon 2020 Program (2014-2020) under grant agreement n° 644571.

The *SHARCS* Consortium

Foundation for Research and Technology – Hellas	Coordinator	Greece
Vrije Universiteit Amsterdam	Principal Contractor	The Netherlands
Chalmers Tekniska Högskola	Principal Contractor	Sweden
Technische Universität Braunschweig	Principal Contractor	Germany
Neurasmus BV	Principal Contractor	The Netherlands
OnApp Limited	Principal Contractor	United Kingdom
IBM - Science and Technology LTD	Principal Contractor	Israel
Elektrobit Automotive GmbH	Principal Contractor	Germany

Document Revisions & Quality Assurance

Internal Reviewers

1. John Thompson & Michail Flouris (ONAPP)
2. Thomas Kamm (Elektrobit)

Revisions

Ver.	Date	By	Overview
1.1	25/12/2015	Elias Athanasopoulos	Release.
1.0	24/12/2015	Elias Athanasopoulos	Finalizing.
1.0b	24/12/2015	Elias Athanasopoulos	Integrated IBM's contribution.
0.9r	24/12/2015	Thomas Kamm	Internal review.
0.9	24/12/2015	John Thompson	Internal review.
0.8	23/12/2015	Christos Strydis	Comments.
0.7	22/12/2015	J. Thomson & M. Flouris (OnApp)	Document review and modifications to cloud application.
0.6	21/12/2015	Michail Flouris (OnApp)	Synced first sections of document from D3.1 for consistency.
0.5	11/12/2015	Elias Athanasopoulos	Pre-liminary release to the QC.
0.4	9/12/2015	Elias Athanasopoulos	Unify presentetion with reporting of WP3.
0.3	16/11/2015	Manolis Stammatogiannakis	Structure revised.
0.2	14/10/2015	John Thompshon	Cloud requirements.
0.1	14/10/2015	Manolis Stammatogiannakis	Document structure.

Contents

List of Figures

Acronyms

ECU	Electronic Control Unit. 15–17
IMD	Implantable Medical Device. 13, 14
OS	Operating System. 20
QoS	Quality of Service. 20
SoC	System on Chip. 13, 14
VM	Virtual Machine. 20

1 Introduction

Work-package 4 (WP4) is the core package for designing and implementing security techniques in software. Primarily, WP4 aims at two important goals. The first goal is to offer support for many hardware components that need special modifications in the software layer. Many advanced security defenses cannot be applied transparently to software by just modifying hardware. For a quick and short example consider the micro-architectures discussed in WP3, which offer hardware support for randomized instruction sets or for control-flow graph labeling. All these features can be only leveraged by applications that are designed for this purpose. In most cases a compiler can enable support of the specialized hardware by just utilizing the additional hardware instructions. Sometimes, the Operating System (OS) should be modified, for example for refreshing the key used in randomized instruction sets during a process context-switch. It is evident that these modifications, which are either applied in the compiler or the OS, are not trivial to carry out (changing a modern compiler is quite challenging these days). Therefore we decided to dedicate a significant part of this deliverable in highlighting possible support for hardware features in software.

The second goal of WP4 is to enable software-only solutions where enhanced hardware is not available. The majority of the techniques outlined in this deliverable, and are discussed in Chapter 4, serve this goal. Again, for a quick and short example, consider that there is no microprocessor available that supports control-flow graph labels. Control-flow Integrity (CFI) techniques [1] can be still applied, by just re-writing a binary (or re-compiling its source if available) for labeling the graph using software pseudo-labels in the form of checks expressed using generic hardware instructions and not special ones. Of course, the software-only version is inferior to the hardware-supported one in terms of performance.

[1] We will elaborate more on how these techniques work in later reporting. Here we refer to CFI just as an example.

It is evident that these techniques are not trivial to implement no matter if they work at the binary or source level. Nevertheless, it is important to stress here, that supporting solutions in a software-only approach is equally important with developing hardware-based techniques. This is mainly for two reasons. First, sometimes changing hardware is simply not realistic (for example in the case of the *SHARCS* Cloud application) and sometimes changing the hardware may be a choice (for example in the *SHARCS* automotive application). This strategy is in-line with what is explicitly stated in the project's DoW: key aspect of *SHARCS* is supporting multiple models of operation.

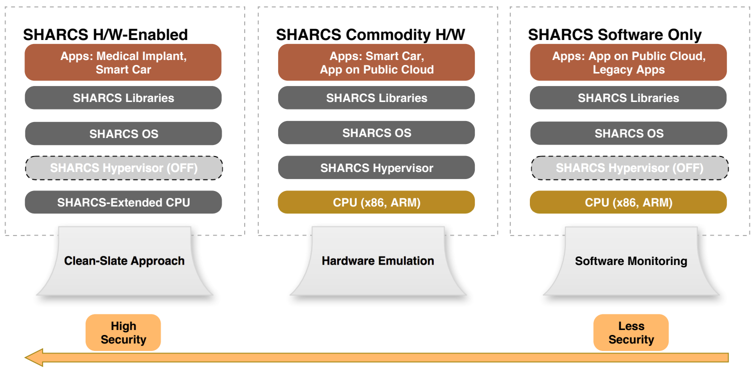

Figure 1.1: Examples of different *SHARCS* framework operational models.

To stress again this multi-model aspect of the project we include all *SHARCS* models as defined in the Description of Work (DoW) in Figure 1.1. Notice, that as stated in the DoW , software changes apply to all different models, either for providing support for hardware-based features or for enabling software-only techniques when custom hardware is not an option. For completeness, we provide here the relevant part from the DoW:

Ideally, SHARCS pushes new functionality to the hardware level, and provides all necessary software-stack changes for producing and running hardened applications. However, modifying all levels is not always possible, therefore we provide two more relaxed *SHARCS-supported models. First, one that incorporates zero hardware changes. For realizing this model all SHARCS features are communicated to a commodity processor using a hypervisor. Second, one that incorporates zero SHARCS features implemented at the CPU, and there is no hypervisor available. For realizing this model, we link the application with SHARCS libraries and add kernel modules at the OS, which embed code for reliably and securely monitoring the application at run-time.*

Finally, the deliverable is concentrated on providing security for three specific applications (Implantable medical device, Automotive, Cloud computing). We stress here two important points. First, there is a significant difference in terms of security requirements from application to application. We have extensively discussed this in D2.1. Second, each application can be potentially configured with multiple *SHARCS* models. For example, the medical implant can be equipped with a sophisticated micro-processor, which supports control-flow graph labeling, but this does not mean that there are no software-only alternatives for similar implants that lack this particular micro-processor.

This deliverable is the main result of task T4.1. The organization of the deliverable is as follows. We first give a short overview of the system architecture of the three different *SHARCS* applications in Chapter 2. These descriptions are partially taken from D2.1. For the full system descriptions the reviewer should refer to D2.1. In the current deliverable we include only the parts needed for making D4.1 self-contained and eliminating cross-deliverable referencing. Subsequently, in Chapter 3 the security requirements of each application to be supported in software, and hence to be addressed in this work-package, are identified. Finally, the document describes in Chapter 4 our current progress and future plans for our activities in T4.2, T4.3 and T4.4. This final part describes in short techniques we are developing as part of our research for hardening software. These techniques are going to be applied to the three *SHARCS* applications and are core contributions of WP4. The exact mechanics of the techniques and their actual integration with the three *SHARCS* applications is the content of future reporting. Finally, in Appendix A we provide a table for summarizing the software techniques used for addressing the specific requirements of the three *SHARCS* applications as discussed in Chapter 3.

2 System Architectures

This chapter is based on the applications' descriptions as discussed in D2.1 and it is included for completeness, easier reading, and eliminating cross-deliverable referencing. The material of this chapter is related to the system architectures of the three *SHARCS* applications. The reviewer that has gone through D2.1 can safely skip this chapter and use it only as a reference for the rest of the text.

2.1 System Architecture for an Implantable Medical Device

The first *SHARCS* application is a medical implant. In particular, Neurasmus provides a device for detecting seizures. A short description of the implant follows, while for the full overview please refer to D2.1.

The *SHARCS* Implantable Medical Device (IMD) Application is a novel, closed-loop, fully implantable neuromodulator that senses EEG and single-neuron recordings, detects seizures before they manifest, and prevents them through highly selective optogenetic (or electric) stimulation of cerebellar neurons. Although implant functionality is autonomous, the implant must also communicate with the outside world for overall control of the device (e.g. recalibration) as well as for sending patient-monitoring information to the patient, the doctor and so on.

The implant is smart, adaptive and autonomous, but it must also allow for operation under the remote control of an external handheld reader device, e.g. a smartphone. The handheld can perform various operations such as implant control, (re)calibration and data logging.

The neuromodulator is implemented in the Neurasmus System on Chip (SoC), depicted in Figure 2.1. Within *SHARCS*, we set the application

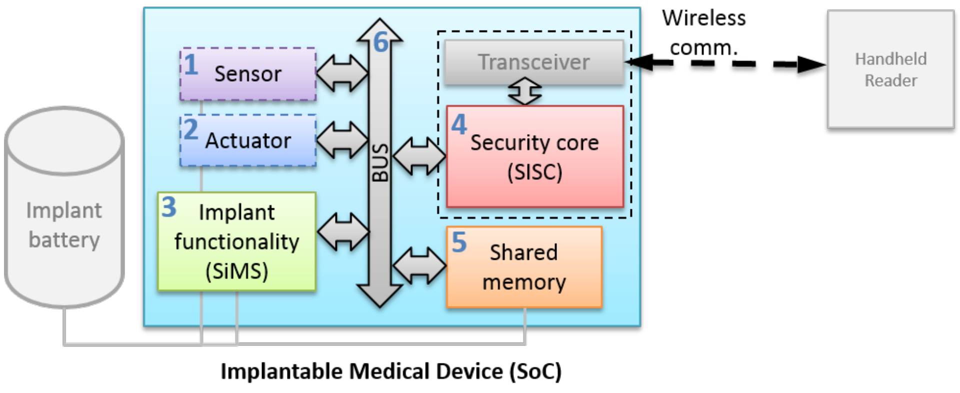

Figure 2.1: The Neurasmus-SoC which implements the implant application.

boundary to contain all digital components plus wireless communication to/from the SoC.

From a software perspective the IMD is really minimal. A single-process application runs which controls the functionality of the device. Although, this is not a full-fledged Operating System (OS), we refer to it with the term OS several times in this deliverable, since it is the software part that operates the device. The minimal software OS can be compiled using a toolchain that supports LLVM bitcode, therefore source-based techniques can be integrated as LLVM phases that modify the bitcode.

2.1.1 Scope

Embedded systems and medical devices have been studied in terms of security. The challenging part in this domain is that vulnerabilities can have severe consequences. In the context of *SHARCS* the medical implant application is interesting, since it gives clearly the opportunity to apply defenses in all available levels. Therefore, security techniques are expected to be applied in this application by modifying both the hardware and software of the device. For software, in particular, certain techniques for leveraging hardware features are in order, like for example compiling software with support for new hardware instructions. In addition, software-only hardening techniques can be applied for cases where support in hardware is not an option.

2.2 System Architecture for Automotive applications

The second *SHARCS* application is an automotive application. In particular, Elektrobit provides software that runs on modern cars and takes care of several functions that can be carried out inside a vehicular environment.

A short description of the system and the software that runs on it follows, while for the full overview please refer to D2.1.

Modern premium cars have up to 80 Electronic Control Units (ECUs). Figure 2.2 shows a selection of different ECUs mounted in a car. Some ECUs, like the engine control and braking system, are essential in driving the car. Other systems such as the airbag are responsible for the safety of the driver. Window lift and Seat control are examples of ECUs controlling comfort functions, which are not necessarily needed to drive the car. The ECUs are interconnected by an on-board network. Furthermore a smart car may communicate with the outside world (e.g. Car-to-Car, Cloud service). Within SHARCS we would like to secure the complete system using a holistic approach.

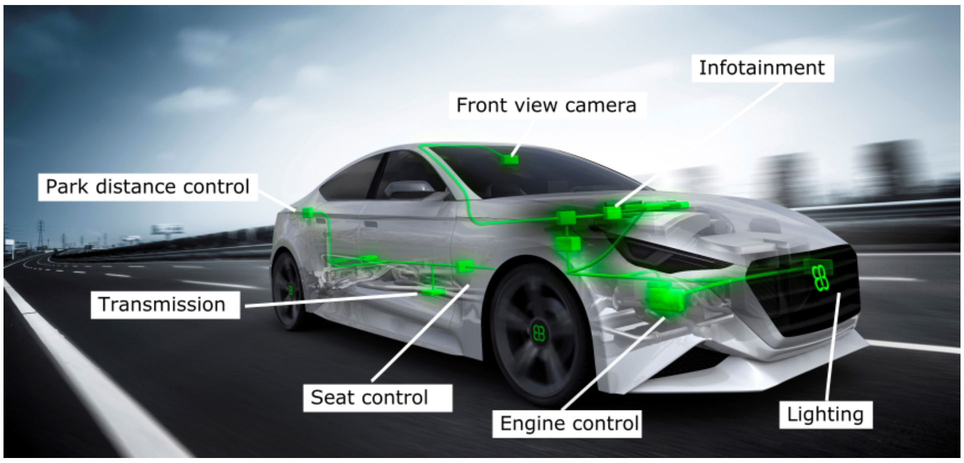

Figure 2.2: Automotive ECU examples.

2.2.1 General description of a typical ECU

An Electronic Control Unit (ECU) is an embedded system with a specific functionality in a car (e.g. engine control, brake). Figure 2.3 shows an abstract overview of an ECU. Depending on the ECU functionality there are different sensors(input) and actuators(output) connected. For example a Heating, Ventilation and Air Conditioning (HVAC) system needs a temperature sensor as an input. Depending on the measured and target temperature the system can control a heater. The ECUs are interconnected by different automotive buses. It is therefore important to secure all ECUs regardless of their function. Otherwise it could be possible for a compromised ECU to gain access to others.

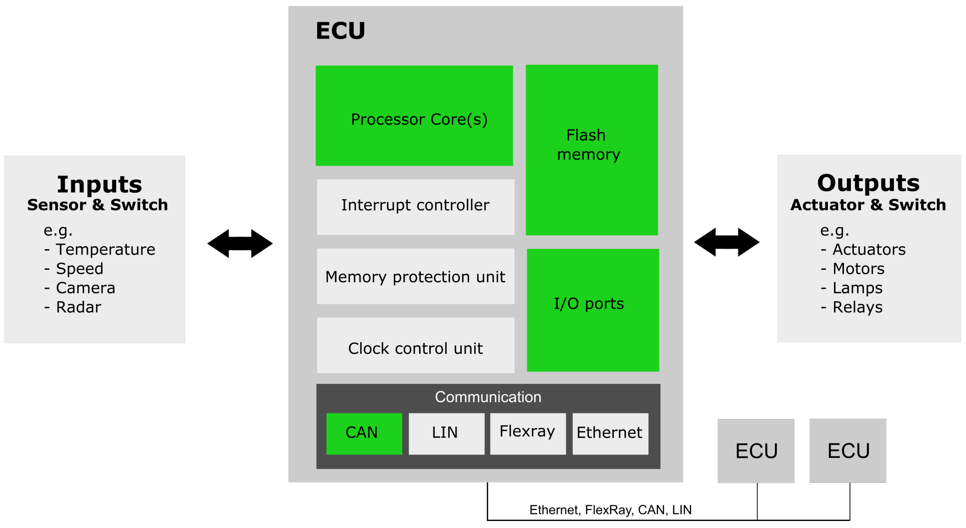

Figure 2.3: Typical ECU.

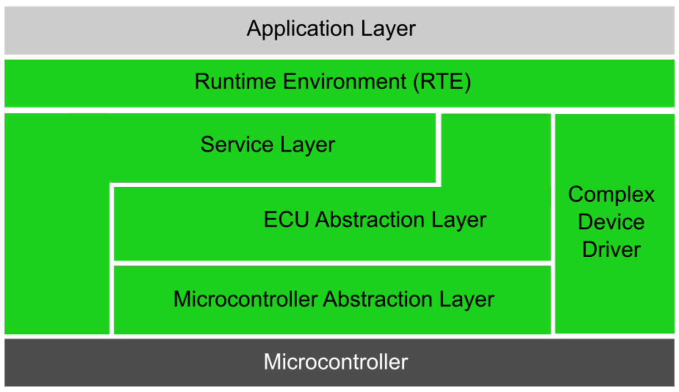

Figure 2.4: AUTOSAR software stack.

In an ECU usually a software application based on the AUTOSAR[1] (AUTomotive Open System ARchitecture) standard is executed on a specific automotive microcontroller (see Figure 2.4). AUTOSAR is a worldwide development partnership of vehicle manufacturers, suppliers and other companies from the electronics, semiconductor and software industry. The layered architecture ensures the decoupling of the functionality from the supporting hardware and software services. Elektrobit supplies a complete software solution, based on AUTOSAR, and provides a software stack in source code, while everything can be adapted/modified in the frame of the *SHARCS* project. See Figure 2.5 for an overview of Eletkrobit's stack. More precisely EB provides:

[1]http://www.autosar.org/

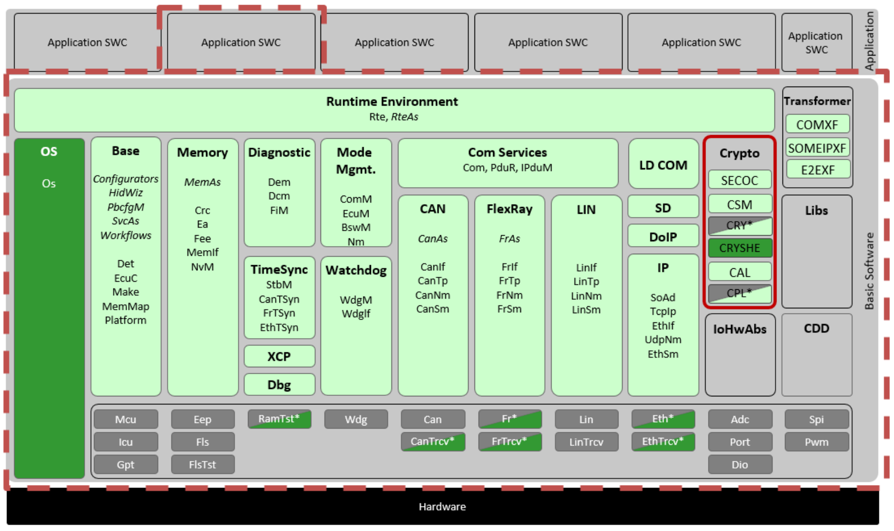

Figure 2.5: AUTOSAR software stack from Elektrobit.

- The AUTOSAR Basic Software (BSW) and Run Time Environment (RTE). This includes for example the operating system, bus communication and memory management.

- An AUTOSAR application consisting of software components (SW-C). In the frame of the SHARCS project this would be a simple demo application to show the effectiveness of the newly added security measures.

2.2.2 General description of a typical car network

All ECUs are interconnected to an on-board network by different automotive buses like CAN, Flexray or Ethernet. The on-board network architecture is different between every car manufacturer and even car model. A network instance is shown in Figure 2.6. All ECUs are usually combined into groups like for example Body Electronics (e.g. Window Lift, Lighting), Infotainment (e.g. Head Unit, Instrument Cluster), Chassis and Safety (e.g. Electric Power Steering, Airbag) and Powertrain(e.g. Engine control, transmission). Communication between the different groups is possible over a gateway. It is therefore important to secure all ECUs as well as the communication between them regardless of their function. Otherwise it could be possible for a compromised ECU to gain access to others.

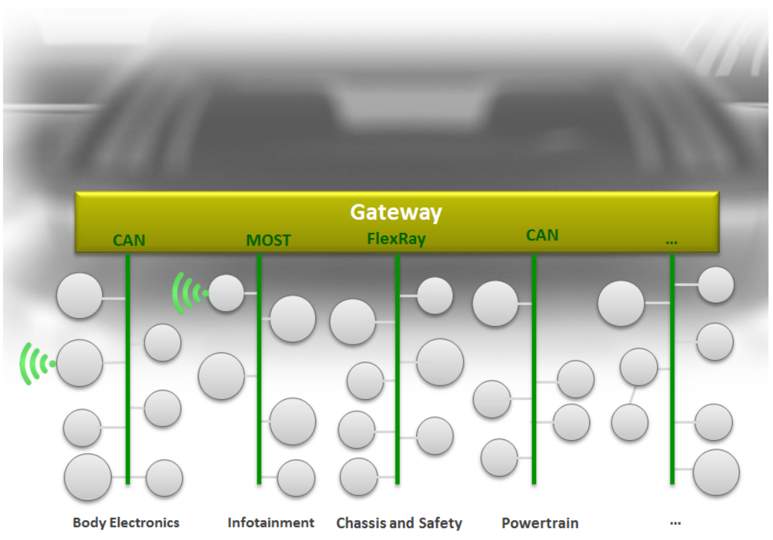

Figure 2.6: Automotive on-board network example

2.2.3 General description of a connected car

Modern cars are more and more connected to the outside world (Figure 2.7). They can communicate with other cars (Car-to-Car), with all kinds of infrastructure (Car-to-Infrastructure), with Cloud Services (e.g. real-time navigation or backup of settings) and with user appliances, such as smart-phones, which can control it remotely. Because of the wireless connectivity a possible vulnerability can be remotely exploited in a large number of vehicles. This must be prevented by security mechanisms against all hazards.

2.2.4 Scope

Embedded security is not entirely new to the automotive domain. Elektrobit provides security mechanisms for more than 15 years to car manufacturers. For several typical automotive use cases partially standardised and many individual solutions exist and are used in vehicles already. The following security use cases are covered today:

- Authentication
- Signature
- Flash protection
- SW-Enabling (OEM-specific or according to HIS)
- Anti-theft mechanisms in SW

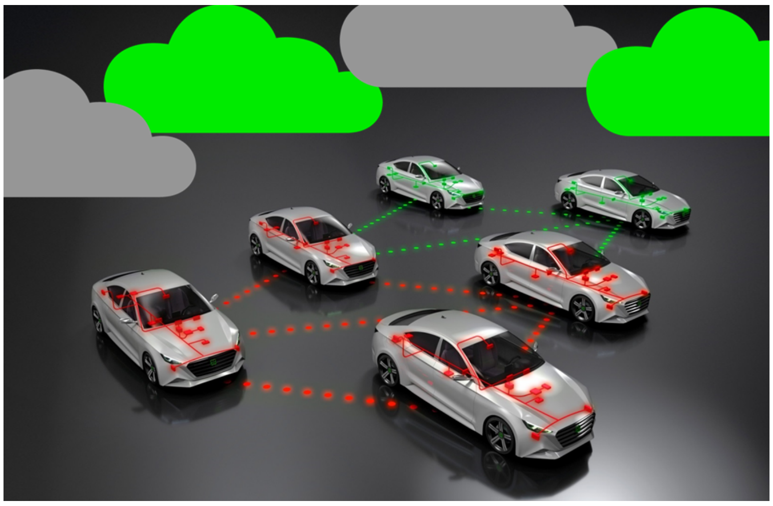

Figure 2.7: Connected car example

- Mileage protection
- Secure Onboard Communication
- Data protection
- Secure Hardware Extension (SHE)
- Hardware Security Module (HSM)
- Secure Boot
- Microkernel OS with memory and execution protection

All of the above usually rely on the use of cryptography for providing security. For the *SHARCS* project one shall assume that the existing mechanisms are (in general) suitable and working as intended. In the context of the *SHARCS* project we are mostly interested in securing the system against exploitation. More precisely, in WP4, we focus on hardening applications either by utilizing hardware support or with software-only techniques.

2.3 System Architecture for Cloud Computing

The final *SHARCS* application is a cloud platform. OnApp builds and provides cloud software based on multiple layers of services. This framework, namely the OnApp Cloud Platform, serves as the cloud application. A short description of the platform follows, while for the full overview please refer to D2.1.

The end-to-end view of the deployment of OnApp Cloud from the repository to the customer Cloud is shown in Figure 2.8. The Cloud platform is used for virtualising hardware resources and presenting those resources to a set of end-users in the form of VM. This helps data center owners to achieve higher utilization of the resources and decouples the management of the virtual machines, which is handled by a company that specializes in cloud hosting. For end-users that just want web-hosting or other types of workload to run as a service, they don't have to be concerned with maintenance, which is taken care of by data center owners.

End-users' Virtual Machines (VMs) can run any type of workload that they would like as if it were a machine that they had full access to. They can choose to use an Operating System (OS) that is in the OnApp template repository or if they would prefer they can prepare a template from scratch. Normally a VM is specialised to perform a single role and will be configured with software applications that allow it to function in that role. The workloads on a VM are normally considered to be the applications that are needed to perform the role with the rest of the system and the other tasks being considered as overhead. From the perspective of a hypervisor server on which these VMs are running the resources are shared as a black box, the workloads and overhead are considered together, when deciding how the resources should be shared between VMs.

The goal of securing this application is to distribute and run in a trusted manner, applications running in a VM on potentially un-trusted (cloud) systems that have limited ability to modify the hardware. To further specify this goal, the owner of VM_1 should be assured that a workload that they are running in VM_1 cannot have its data compromised or its performance affected beyond contracted Quality of Service (QoS) levels. These exploits could be indirectly caused by a secondary VM, VM_2 that may be misconfigured or compromised. In a Cloud environment we must consider that some legacy applications will not have the source code available whereas for some other applications the source code may be available. This applies for the host hypervisor platform, the guest operating system and the workloads.

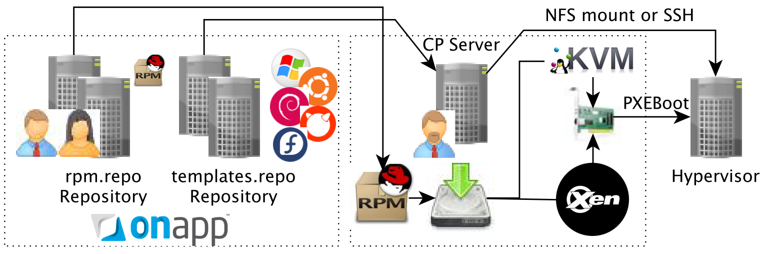

Figure 2.8: End-to-end view of the OnApp platform from the repository through to deployment.

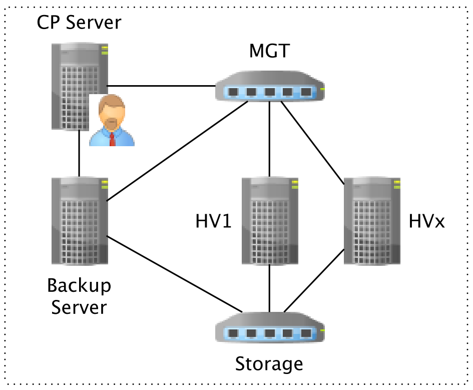

Figure 2.9: OnApp simplified Cloud deployment.

The boundary of the system is considered to be the cloud deployment on the customer site. A minimal cloud deployment includes (as shown in Figure 2.9);

- CP2 server that is the public facing server that manages the cloud deployment;
- a backup server is optional but highly recommended;
- one or more servers (two or more to have migration and other useful functions) that act as hypervisors for hosting virtual machines.

Normally a SAN appliance is required for a cloud deployment to host the storage for the virtual machines. OnApp have a product, Integrated Storage[2], which means that for OnApp Cloud deployments an external SAN is not required. It is therefore assumed for this application that the storage is hosted on the hypervisors themselves and that Integrated Storage is used.

2.3.1 Scope

The cloud environment is significantly rich, a broad variety of security issues can be considered, and defenses can be applied to several layers. Here we clarify the scope of *SHARCS* for the cloud application.

Many of the attacks and vulnerabilities for cloud-based computing systems come from attacks to the cryptographic elements in the platform, attacking weaknesses in the keys, fundamental security flaws, and flaws in the implementation of cryptographic mechanisms. Many projects cover the security of cryptographic components and as such they will not be directly

[2]http://www.onapp.com/storage

addressed by *SHARCS* with a minor exception. In *SHARCS* we protect sensitive data like cryptographic keys in GPUs. Notice, that the nature of the protection is on the systems level, and not on the cryptographic level. Additionally, in the context of *SHARCS* we are interested mostly in software hardening for preventing exploitation. Hardening software can be done by either applying techniques to binaries or to source code. As it has been already mentioned, the cloud environment includes many legacy applications, which cannot be re-compiled. Finally, any changes to the underlying hardware will undermine many of the principles and operations of the cloud platform. Given that data centers should normally have strict access controls to the hardware we will work on the basis that hardware cannot be modified and physical attacks are out of scope. Therefore, all *SHARCS* techniques for the cloud application should be considered software-only.

3 Application requirements that need software support

In this part we discuss how all security requirements of the three *SHARCS* applications are addressed in software. The methodology and the product for identifying the security requirements are part of D2.1. This Chapter primarily aims at fulfilling two goals. The first goal is to discuss how support of hardware components that are discussed in WP3 reaches the software layer, through particular APIs, toolchains and OS support. The second goal is to discuss software-only techniques, in cases where hardware support is not available.

We begin with a generic overview of the benefits stemming from supporting security in software;

- Utilization of particular hardware features that are not designed in principle for offering security;
- Selectively leverage hardware features for applying security techniques in particular applications;
- Application of security defenses that are based on hardware features to architectures that do not support the related features;
- Deployment of recent security technologies to legacy systems;
- Flexibility in applying techniques in different layers (OS, hypervisor, compiler, application, etc.);
- Faster deployment, since changing software is easier that changing hardware;
- Easier patching of faulty security implementations via software.

There are also disadvantages of using software-based defenses:

- Increased overheads when hardware support is absent;

- Complexity due to interference of multiple components;
- Lack of transparency;

In the following part of this chapter we review the requirements of each application, which, in the context of *SHARCS*, receive support in software by other complementing hardware features or by offering software-only replacements.

3.1 Software Requirements for Implantable Medical Devices

The application evaluation in *SHARCS* deliverable D2.1 has identified a number of security requirements that apply, which we repeat here for convenience:

SR-1.1 Security compliance with extra-functional constraints.
Briefly stated, these extra-functional constraints are:

(a) Real-time execution of the IMD functionality (10 ms);

(b) Increase power and energy consumption by no more than 10%;

(c) Increase device-area by no more than 30%.

SR-1.2 Security compliance with proper treatment delivery;

SR-1.3 Patient-data security and privacy;

SR-1.4 Patient safety & device accessibility;

SR-1.5 Security compliance with maintenance tasks.

The medical implant is an example application that follows the clean-slate *SHARCS* model, since all hardware can be modified, and offering security at the hardware level is typically more efficient (performance, energy) than software-based solutions. Additionally, all hardware technologies must be exposed to the software running in the device. For example the software of the device should be annotated for running on a microprocessor that supports machine instructions for CFI or accelerated cryptographic operations. Therefore, in this part we highlight the software components that promote hardware features to the application. Additionally, we provide purely software-based techniques that do not rely on particular hardware components. Therefore, multiple *SHARCS* models apply to this application, which can be configured accordingly.

SFR-1.1 **Cryptographic operations (SR-1.1)** Annotate software for supporting the lightweight cryptographic operations of the SISC core in the Neurasmus SoC. Software running in the device can take advantage of the fast cryptographic operations implemented in hardware.

SFR-1.2 **Prevent code injection and code reuse (SR-1.2 through SR-1.5):** Library and compiler toolchain for supporting hardware-based ISR and CFI. The hardware architectures described in WP3 for executing programs using a randomized instruction set (ISR) and for adhering to their control-flow graph (CFI) need special OS and compiler support. Programs should be compiled for leveraging the new architectures, and, in the general case, the OS should be modified accordingly. For the medical implant the OS is contained in a single-process application running on the bare metal, therefore producing a CFI/ISR-enabled binary is sufficient. In WP4 we provide all the necessary software stack for enabling the hardware architectures to the software layer. Additionally, we provide software-only hardening techniques that support secure program execution. Applications are compiled with a custom toolchain that inserts checks in all indirect branches. For more details about how these techniques work please refer to Chapter 4.

SFR-1.3 **Prevent data leakage (SR-1.3):** Library for supporting the secure and encrypted IMD communication stack. The library is provided in two versions. One version is based on the cryptographic library that supports the accelerated instructions available in hardware, and the other one is based on the equivalent software-based library, which supports standard cryptographic operations.

SFR-1.4 **Enable emergency authentication (SR-1.4):** Implement support for the emergency authentication mode in the software of the IMD application.

3.2 Software Requirements for Automotive applications

The application evaluation in *SHARCS* deliverable D2.1 has identified a number of security requirements that apply, which we repeat here for convenience:

SR-2.1 **Message manipulation:** An attacker shall not be able to impersonate another sender of messages which are received by the controller on a communication bus in such a way that the controller executes code which the attacker provides.

SR-2.2 Data flash manipulation: If there is a data flash module on the controller which can be manipulated by an attacker, the attacker shall not be able to manipulate the data flash in such a way that the controller executes code which the attacker provides.

SR-2.3 Single controller execution: If the attacker is able to manipulate a single controller in such a way that the controller executes code which the attacker provides, the method used for this controller shall not be possible on a different controller running the same software stack.

SR-2.4 Software module isolation: If an attacker is able to modify the source code of one of the basic software modules or the application modules, the attacker shall not be able to obtain information about data in the other basic software modules and application modules.

The Automotive Application is very security-critical and the on-board and off-board vehicle communication must be enhanced with security mechanisms to prevent attackers from manipulating functionality or gaining unauthorized access to those ECUs. Such protection is essential for the safety and security of passengers on the road.

All hardware technologies must be exposed to the software running in the device. Therefore, in this part we highlight the software components that promote hardware features to the automotive application. Additionally, even though hardware-based security mechanisms have the highest standard it is not always feasible to adapt the hardware (e.g. off the shelf automotive microcontroller). In this case software security mechanisms are appropriate.

3.2.1 Software security requirements

SFR-2.1 Code injection (SR-2.2 through SR-2.3) Library and compiler toolchain for supporting hardware-based ISR. We thoroughly describe the architecture of a microprocessor that supports Instruction Set Randomization (ISR) in WP3. This microprocessor prevents any code injection of running applications, as long as the randomization key is kept secret and safe. For executing ISR-enabled programs the OS should be modified accordingly (e.g., the loader should be able to run randomized binaries, and the kernel should be modified to save the key during context-switch). In WP4 we will provide all the necessary software stack for enabling the ISR-enabled applications. Additionally, we provide software-only hardening techniques that allow for secure program execution. For this particular case, we apply a technique that randomizes the stack and prevents exploitation of stack-based vulnerabilities. For more details about how these techniques work please refer to Chapter 4.

SFR-2.2 **Code reuse (SR-2.2 through SR-2.3)** Library and compiler toolchain for supporting Control-flow Integrity (CFI) on embedded systems. We thoroughly describe the architecture of a microprocessor that supports CFI in WP3. This microprocessor prevents exploits based on code reuse, since the running code should adhere to the control-flow graph. For executing programs that are CFI-enabled, the code should be annotated with the new CFI instructions, while the OS should be modified accordingly (e.g., handle violations of the CFI policy). In WP4 we will provide all the necessary software stack for enabling the CFI-enabled applications. Additionally, we provide software-only hardening techniques that provide secure program execution. For this particular case, we apply coarse-grained CFI to legacy binaries, and fine-grained CFI to applications of the automotive stack that can be recompiled. For more details about how these techniques work please refer to Chapter 4.

SFR-2.3 **Encryption of shared memory data (SR-2.4)** Toolchain for supporting hardware encryption of shared memory data. Additionally, we will provide software-based only cryptographic libraries.

3.2.2 Software application requirements

- **Interoperability:** Operating system and compiler tool chain should allow modifications.
- **Compatibility:** Binaries format should be open (e.g., ELF).
- **Performance:** The software security mechanisms should only have minimal impact on the execution time.
- **Power:** Power consumption overhead should be as little as possible but is not as critical as in the implant application.

3.3 Cloud application

The application evaluation in *SHARCS* deliverable D2.1 has identified a number of security requirements that apply, which we repeat here for convenience:

SR-3.1 **End-user security and privacy**

SR-3.2 **Integrity of the platform and workloads**

SR-3.3 **Availability of the platform**

SR-3.4 **All operations must be attributable to a user**

SR-3.5 All operations must be authenticated

In D2.1 the security threats related to the Cloud application domain have been enumerated and in D2.2 further investigation into the attack vectors and specific vulnerabilities was performed. Some of the risks can be mitigated through a combination of software and hardware techniques. In WP3 we thoroughly discuss defenses implemented in hardware, while here we elaborate more on the software side. Notice, that the power utilization and energy efficiency for security solutions in the data center is not as critical as it is for the Automotive or IMD applications. Therefore, most techniques for the Cloud application can be software-only. Additionally, hardware-based defenses can be provided to applications running in the Cloud through software emulation.

3.3.1 Security requirements - software

SFR-3.1 Software API for all hardware requirements discussed in D3.1 (SR-3.1 through SR-3.5): In D3.1 we thoroughly discuss a number of hardware components for addressing all security requirements of the cloud application, as defined in D2.1. Additional hardware components and features require software support for enabling techniques in applications. This is usually enabled through a software Application Programming Interface (API).

SFR-3.2 Software hardening (SR-3.1 through SR-3.3): Advanced code-reuse attacks may affect the integrity and the availability of the cloud platform and lead to application compromising. Software hardening techniques will be applied for strengthening the cloud environment against software exploitation. The techniques can be applied directly to binaries, for legacy applications, or by re-compiling software. The goal of the techniques is to prevent exploits from happening, even when vulnerabilities are in place. Consider, that the cloud environment is rich, and providing solutions for vulnerability-free software is not realistic in this case. Therefore, all defenses focus on preventing exploitation, rather than correcting bugs.

SFR-3.3 Advanced Authentication (SR-3.4 through SR-3.5): Authentication will be hardened in both client and server side. For the server side, software that leverages GPU storage for shielding sensitive cryptographic keys will be used. This software protects the server's certificate from compromising.

3.3.2 Software application, beneficial features

Additionally to software support for particular hardware components, the Cloud application utilizes many sophisticated software-only techniques. These technologies can be applied to both closed-source legacy applications by directly modifying binaries or to open-source applications by re-compiling them. All technologies delivered by WP4 for the Cloud application are based on the following techniques.

- **Binary re-writing engines:** Certain frameworks exist for de-composing, analyzing, and hardening binaries, as for example DynInst [6].
- **Compiler extensions:** In cases where source is available, software hardening techniques can be incorporated in the compiler (e.g., by adding an LLVM phase).
- **Dynamic Binary Instrumentation (DBI):** In cases where source is not available and the binary cannot be statically analyzed dynamic instrumentation will be used using Intel's PIN framework [12].

3.3.3 Application software requirements

- Operations. The operating system should be based on the Linux kernel.
- Operations. The compiler(s) used should be based on LLVM/Clang or GCC.
- Compatibility. The operating system should support standard virtualization features.
- Compatibility. The operating system should support dynamic linking.
- Business. Software hardening techniques should be open.
- Business. Additional hardware components for authentication (e.g., GPUs) should be based on commodity models.
- Business. All virtualization support would be delivered through Xen or KVM.

3.3.4 Software assumptions

- No malicious software installed before applying the techniques; a safe bootstrap phase of the operating system is assumed.
- Operating system is trusted and, if not compromised, does not maliciously interfere with the rest of the applications.

- Hardware is trusted.

4 Current progress and future plans on software security mechanisms

In this part we discuss the progress of research related to the techniques outlined in this deliverable. As already stated this deliverable has mainly two goals. The first goal is to discuss how support of hardware components reaches the software layer, through particular APIs, toolchains and OS support, and, the second goal is to discuss software-only techniques, in cases where hardware support is not available. In this part we discuss some representative software-only techniques for hardening software. The requirements that serve the first goal are listed in Table 4.1 and the requirements that serve the second goal are listed in Table 4.2. Notice that **SFR-1.2** is included in both lists, since for this particular case we enable multiple *SHARCS* models.

In this section we discuss the progress we have so far in developing techniques that target the requirements listed in Table 4.2. Some of the techniques that work with binaries can be applied to legacy components of *all* applications, while some others offer protection only when source is available and the protected software can be recompiled. The techniques that work on binaries are PathArmor and StackArmor, which are presented in Sections 4.2 and 4.3 and target requirements **SFR-2.2** and **SFR-3.2**. When

H/W Support
SFR-1.1 Cryptographic operations
SFR-1.2 Prevent code injection and code reuse
SFR-1.3 Prevent data leakage
SFR-1.4 Enable emergency authentication
SFR-3.1 Software API

Table 4.1: List of requirements where changes of the *SHARCS* software stack are needed.

Software-only
SFR-1.2 Prevent code injection and code reuse
SFR-2.1 Code injection
SFR-2.2 Code reuse
SFR-3.2 Software hardening
SFR-3.3 Advanced Authentication

Table 4.2: List of requirements where *SHARCS* software provides security defenses without H/W support.

source code is available then ShrinkWrap, which is presented in Section 4.1 can be applied (**SFR-1.2**, **SFR-2.1**, **SFR-2.2**, and **SFR-3.2**).

In Section 4.4 we present a software application that, partially, runs on a GPU and is able to shield sensitive data, like cryptographic keys, in the memory of the graphics card. This approach has several advantages as we discuss later in the section. Notice, that the GPU involved can be commodity without incorporating particular custom features. This application is addressing **SFR-3.3**. Finally, in Section 4.5 we report on a custom software developed by IBM for vulnerability finding, which will assist in covering **SFR-3.2**.

We summarize all techniques and requirements in Appendix A.

4.1 Source-based VTable protection

ShrinkWrap is a compiler extension for hardening C++ source code. Primarily ShrinkWrap targets C++ VTables, which is essentially a hard to protect form of function pointers, but the system can also provide protection of other elements that lead to indirect branches (such as return addresses and C function pointers). In this section we elaborate on the C++ protection, since it is the most interesting and challenging. Nevertheless, the technique should not be considered that targets C++ source code only; generic C source code is supported, as well.

C++ is a popular, fast, object-oriented (OO) language used to develop some of the most popular Web browsers, including Chrome and Mozilla. Due to their popularity, size and complexity, applications developed in C++ are frequently targeted by attackers. Despite advances in software security, like the introduction of data-execution prevention [4], stack-smashing protection [8], and address-space layout randomization [16], their exploitation is still possible. New techniques involving information leaks [21] and return-oriented programming [20] are employed to bypass protection mechanisms and perform arbitrary code execution attacks.

One of the features of C++ applications targeted by attackers are *virtual function tables*, or *VTables*. OO languages support run-time method binding, i.e., determining the method to be called based on the run-time type of an object, instead of the static type of the pointer pointing to that object. Modern compilers typically provide this functionality through *VTables*, which provide an efficient way to call the correct method at run time. Unfortunately, VTables are based on indirect calls, which is what makes them a prominent targets for hijacking the control flow of a program.

To prevent such control-hijacking attacks, the research community has turned to *control-flow integrity* (CFI). First conceived in 2005 [3], CFI has seen a long line of followers and variants since [24, 25, 14]. CFI strives to constrain the control flow of a program to its statically-determined control-flow graph (CFG) as strictly as possible. In principle, CFI can be very effective in preventing a wide-range of attacks. In recent times however, we bear witness to a cat-and-mouse game, where each new CFI technique is immediately attacked and bypassed. Earlier works have shown that attackers can bypass loose CFI mechanisms [10], so follow-up works tried to exploit source code information [14] and VTables semantics [13, 7, 5] to make CFI more fine-grained. A very recent work has shown that the above approaches still leave programs vulnerable and argue that unless you correctly extract C++ semantics from source code they will remain vulnerable in the future [19].

We have investigated in detail current protection systems [23]. Our investigation exposes *three* weaknesses of existing schemes. First, we have found that existing solutions fail to precisely identify the object types associated with a virtual call-site, even in the presence of source code. Second, we have found that even state-of-the-art solutions, like VTV [23] which is developed by Google, handle multiple inheritance over-permissively. Normally, every class has its own VTable and base classes contain all the VTables of their subclasses. When a class C inherits from multiple classes, VTV extends the VTables of its base classes to include, and thus share, all entries in their individual VTables. I.e., "sibling" classes share VTable entries. This is another example where control-flow integrity is loosely enforced. Finally, we have identified a fundamental error in the assumptions made by other solutions. Previous approaches operate on the premise that allowable control-flow transfers at call-sites (i.e., where a method of an object is invoked) can be determined solely based on the type of the object pointer involved. We show that this assumption is false and more information must be extracted from the call-site to reach optimal protection.

Based on these observations, in VU, we have designed a new VTable-protection scheme that uses information available during compilation to extract the most restrictive set of VTables that should be accessible at a virtual call-site within the code. Our progress at this point, can be summarized as follows:

- We have identified limitations in the design and implementation of current VTable protection schemes, including the primary industrial implementation by Google, VTV [23].

- We have identified key design decisions that should be accounted for, when dealing with VTable protection. This also includes a definition of optimal (minimal) VTables sets that should be accessible at each point in the program.

- We have developed a practical testing methodology to evaluate VTable protection schemes and to highlight potential limitations.

- We have implemented a prototype of the proposed protection scheme and have evaluated it on a large, complex real-world application, the Chrome, browser, in terms of security *and* speed.

4.2 Advanced CFI for binaries

A key problem with traditional CFI solutions—even recent source-level fine-grained ones [23]—is that they enforce only context-insensitive CFI policies, which examine control edges in isolation and attempt to statically derive the resulting superset of all the possible targets according to the CFG. The lack of context inevitably results in weak CFI invariants, allowing attackers to freely chain edges together and form paths that are even trivially infeasible in the original CFG (e.g., returning to a function never on the active call stack [10]).

Context-sensitive CFI techniques are a promising way to address this problem, since they rely on context-sensitive static analysis to associate CFI invariants to *paths*—i.e., multiple consecutive edges—in the CFG and enforce such invariants on execution paths at runtime. The stronger security guarantees provided by context-sensitive CFI techniques have been acknowledged as early as in the original CFI proposal, but their real-world adoption has been rapidly dismissed as impractical [3].

In our research, we have demonstrated that Context-sensitive CFI (CCFI) can indeed be implemented in an efficient, reliable, and practical way for real-world applications. We have developed *PathArmor*[1], a binary-level CCFI solution which enforces context-sensitive CFI policies on both the backward and forward edges. *PathArmor* relies on commodity hardware support to efficiently and reliably monitor execution paths to sensitive functions which

[1]*PathArmor* is open source, available via https://github.com/dennisaa/patharmor

can be used to mount control-flow diversion attacks [15], and uses a carefully optimized binary instrumentation design to enforce CCFI invariants on the monitored paths. *PathArmor*'s path invariants are derived by a scalable context-sensitive static analysis performed over the CFG on-demand, which uses caching of path verification steps to achieve high efficiency. Verification itself is also very efficient, since all the CFI checks are batched at sensitive program points.

To show the practicality of our design, we have prototyped two context-sensitive and binary-level CFI policies (for the backward and forward edges, respectively) on top of *PathArmor*. Moreover, our framework can also serve as a general foundation for even stronger CCFI implementations, for instance using context-sensitive data-flow analysis at the source level. Even in the current setup, *PathArmor* provides a comprehensive CCFI protection system with much stronger security guarantees than traditional CFI solutions, while matching or even improving their performance. Moreover, due to its optimized design, *PathArmor* can also serve as an efficient basis for fine-grained context-insensitive CFI ($\overline{\text{C}}$CFI) policies.

In short, our progress can be summarized as:

- We have identified the key challenges towards practical CCFI implementations and investigate opportunities to address these challenges in real-world applications and commodity platforms.

- We have developed *PathArmor*, a framework to efficiently support arbitrary context-sensitive and context-insensitive CFI policies on commodity platforms. To fulfill its goals, *PathArmor* relies on hardware support, binary instrumentation, and on-demand static analysis to batch even sophisticated CFI checks at the relevant sensitive points in a binary.

- We have evaluated *PathArmor* using popular server applications and the SPEC CPU2006 benchmarks. Our results show that *PathArmor* can significantly restrict the number of legal control flows compared to traditional CFI solutions (-70% across all our applications, geometric mean), while yielding bounded memory usage (+18-74 MB on our applications) and low run-time performance overhead (3% on SPEC and 8.4% on our applications, geometric mean).

4.3 Stack protection for binaries

Information leakage and buffer overflow attacks, in particular, are greatly helped by the predictability of the stack layout. Although the start is typically randomized, the stack itself grows in an entirely predictable fashion, making the disclosure of canaries, return addresses, or data pointers of previous stack frames as simple as leaking uninitialized data or exploiting buffer overreads. The same applies to exploits modifying data in another stack frame. For example, randomization between stack can stop many high-profile attacks.

Our focus on binaries is neither academic nor fundamental, but important in practice: the adoption of advanced security measures in popular compilers is slow. Compiler maintainers are conservative and wont to reject options that incur significant overhead. The `-fstack-protector-strong` option in `gcc` is a case in point: it had to be tailored to a very narrow threat model for performance reasons. `gcc`, any measure not added to it for performance reasons will not make it into their products. Unless they apply the defenses at the binary level, users cannot decide for themselves to sacrifice some performance for better security. Additionally, many *SHARCS* applications contain, at least in part, legacy software that cannot be recompiled.

In *SHARCS* we apply a novel stack-protection technique that shields binaries from exploits based on stack vulnerabilities. To provide comprehensive protection, we rely on static analysis enabled by state-of-the-art binary analysis tools—which provide the necessary program abstractions, such as functions and their control-flow graphs. Our static analysis is also supported by information on the location and size of stack objects, for example provided by debug symbols.

Using binary rewriting to instrument call and return instructions, we provide tailored protection based on application-specific performance and security requirements. We rely on a combination of randomization, isolation, and secure allocation techniques to create the illusion that *all* the stack frames and the individual stack buffers are drawn from a fully randomized space with no spatial or temporal predictability guarantees. Unlike all the existing solutions, this strategy can comprehensively protect against *arbitrary* stack-based attacks.

In short, our progress can be summarized as:

- We have developed StackArmor a novel stack-protection technique which combines inter- and intra-frame defenses to stop arbitrary spatial and temporal attacks.
- We have implemented the technique in x86_64 Linux, therefore can be easily applied to the Cloud environment, when software re-compilations is not possible.
- Through a detailed experimental evaluation of our prototype, we have concluded that it achieves a modest performance and physical memory overhead of 5% and +3 MB, respectively, on average, on single-threaded server programs, while scaling well even to heavily threaded

server programs (28% and +112 MB with 100 worker threads, on average) with full protection.

4.4 Safe storage using GPUs

Servers have always been an attractive target for attackers, especially when they host popular web sites and online services, as they typically contain a wealth of private user data and other sensitive information. Encryption can be used as an additional layer of protection for sensitive data, once a service has been compromised, but it is only effective as long as the keys involved in cryptographic operations are kept secret. In fact, keys themselves are often the target, as for example is the case with the infamous Heartbleed bug [1]. The exploitation of Heartbleed, a buffer over-read vulnerability in OpenSSL, allows attackers to read arbitrary contents from a server's memory, including TLS private keys. Besides attacks that leverage software vulnerabilities to disclose memory or take complete control of the host, key recovery attacks can also be mounted using direct memory access through Firewire [17] or PCI [22]. Moving one step further, it has been demonstrated that keys can be extracted by freezing memory chips and inspecting their contents [11].

Once the secret keys are leaked, attackers can impersonate the server (without triggering any browser warnings), or decrypt any past and future captured encrypted data (unless perfect forward secrecy is used). Defenses that involve the in-memory obfuscation of keys using dispersal techniques offer only partial protection, as attackers can eventually break the obfuscation scheme with adequate effort. To that end, it is crucial that, apart from the trusted operation of the underlying cryptographic implementation, secret keys and other sensitive information is safely stored and protected from leakage. It is important to ensure that a potential security flaw in a service will not allow an adversary to get access to secret keys, even if the service is fully compromised, as this can lead to further catastrophic consequences [2, 1].

We explore an alternative approach to the problem of protecting a server's cryptographic keys, which takes advantage of the graphics card to exclusively i) store cryptographic keys and other sensitive information, and ii) carry out all cryptographic operations, without involving the CPU. Our prototype system, named *PixelVault*, provides native GPU implementations of the AES [9] and RSA [18] algorithms, and prevents key leakage even when the base system is *fully* compromised. This is possible by exposing private keys only in GPU registers, and keeping PixelVault critical code exclusively in the GPU instruction cache, preventing this way even privileged host code from accessing any sensitive code or data.

We have implemented a PixelVault-enabled version of the OpenSSL library, which allows the transparent protection of existing services without hardware modifications or operating system recompilation. Multiple services can use the same GPU to perform cryptographic operations, using the same or different certificates (and secret keys), while trust is always given to a single hardware entity—the GPU.

Our choice of the GPU for key storage is justified by its unique properties, including *(i) non-preemptiveness:* all program code running on the GPU is never context-switched, and therefore, there is no saved state in the host's memory that could include information associated with cryptographic keys; *(ii) on-chip memory operation only:* the running GPU code is tamper-resistant in on-chip memory, and the associated cryptographic keys are never stored in observable memory, but only in non-addressable memory, such as the registers of the GPU; *(iii) transparency:* the GPU is independent from the host, so no hardware, operating system, or application changes are required—just a modification of the standard cryptographic libraries used, such as OpenSSL, which essentially implies that legacy applications can fully take advantage of our system with minimal effort; *(iv) commodity component:* GPUs are commodity components and are cheaper than dedicated cryptographic hardware; *(v) performance:* GPUs achieve high computational performance for cryptographic operations, for applications in which they can be parallelized.

In short, our progress can be summarized as:

1. We have designed a system for keeping cryptographic keys and carrying out cryptographic operations exclusively on the GPU, which allows it to protect secret keys from leakage even in case the host is fully compromised.

2. We have implemented the system using commodity GPUs (NVIDIA's GTX 480), and provide a compatible version of the OpenSSL library.

3. Our system does not only provide better protection, but also outperforms CPU-based solutions in terms of processing throughput for server applications.

4.5 Source-code screening

SHARCS will use offline source code screening tool, capable of finding security vulnerabilities in the code, as well as proving to some extent that some of the code components are clean of specific types of vulnerabilities (e.g. memory safety issues). The benefit *SHARCS* platform gains from the tool is as follows: components that were successfully screened will be considered ”clean”, leading to better utilization of *SHARCS* hardware and software online technologies for only ”dirty” components, improving the overall performance, memory utilization and other resource consumption metrics of the whole system. The tool will be provided under the Task 4.3.

A

Mapping of Software Techniques to Software Requirements

Software Requirement	H/W support	Source CFI	Bin CFI	Stack Rnd	GPUs	Screening
SFR-1.1 Cryptographic operations	✓					
SFR-1.2 Prevent code injection and code reuse	✓	✓				
SFR-1.3 Prevent data leakage	✓					
SFR-1.4 Enable emergency authentication	✓					
SFR-2.1 Code injection		✓				
SFR-2.2 Code reuse		✓	✓	✓		
SFR-3.1 Software API	✓					
SFR-3.2 Software hardening		✓	✓	✓		✓
SFR-3.3 Advanced Authentication					✓	

Table A.1: Software requirements and techniques.

Bibliography

[1] The Heartbleed Bug. http://heartbleed.com/.

[2] Who holds the encryption keys? http://www.computerworld.com/s/article/9225414/Who_Holds_the_Keys_.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proc. of the 12th ACM CCS*, 2005.

[4] S. Andersen and V. Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention, 2004. http://technet.microsoft.com/en-us/library/bb457155.aspx.

[5] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proc. of the 22nd NDSS*, 2015.

[6] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *PASTE*, 2011.

[7] Chao Zhang, Chengyu Songz, Kevin Zhijie Chen, Zhaofeng Cheny, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proc. of the 22nd NDSS*, 2015.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of Usenix Security'98*.

[9] J. Daemen and V. Rijmen. AES Proposal: Rijndael, 1998.

[10] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *Proc. of the 35th IEEE S&P*. IEEE, 2014.

[11] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest We Remember: Cold-boot Attacks on Encryption Keys. *Communications of the ACM*, 52(5):91–98, 2009.

[12] Intel. Pin - A Dynamic Binary Instrumentation Tool. http://www.pintool.org/, 2011.

[13] D. Jang, Z. Tatlock, and S. Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Proc. of the 21st NDSS*, 2014.

[14] B. Niu and G. Tan. Modular Control-flow Integrity. In *Proc. of the 35th ACM PLDI*, 2014.

[15] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proc. of Usenix Security'13*.

[16] PaX Team. Address Space Layout Randomization (ASLR), 2003. pax.grsecurity.net/docs/aslr.txt.

[17] D. R. Piegdon and L. Pimenidis. Targeting Physically Addressable Memory. In *Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA, 2007.

[18] R. L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of ACM*, 21, February 1978.

[19] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proc. of the 36th IEEE S&P*, May 2015.

[20] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proc. of the 14th ACM CCS*, 2007.

[21] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proc. of the 34th IEEE S&P*, 2013.

[22] P. Stewin and I. Bystrov. Understanding DMA Malware. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA, 2013.

[23] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-edge Control-flow Integrity in GCC and LLVM. In *Proc. of Usenix Security'14*.

[24] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proc. of the 34th IEEE S&P*, 2013.

[25] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proc. of Usenix Security'13*.