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 D4.2: Design specification of the SHARCS runtime system, software tools and reporting

Abstract: This deliverable describes the core design specification of the SHARCS software technologies. It first surveys the classes of attacks and vulnerabilities SHARCS software technologies seek to defend against, based on the requirements analysis in D2.1. It then illustrates our security approach to counter such vulnerabilities, with a portfolio of several techniques at different design layers, ranging from software verification to integrity monitoring and recovery. Finally, it provides a design overview of all the software SHARCS technologies, positioning each technology on a target application domain with its own implementation constraints (based on the applications in WP5) and within a particular design layer.

Contractual Date of Delivery	Month 24
Actual Date of Delivery	Month 24
Deliverable Dissemination Level	Public
Editor	Cristiano Giuffrida
Contributors	All *SHARCS* partners
Quality Assurance	John Thomson (OnApp), Vassilis Prevelakis (TUBS)

† 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 Thomson (OnApp)
2. Vassilis Prevelakis (TUBS)

Revisions

Ver.	Date	By	Overview
0.7	24/12/2016	Cristiano Giuffrida	Addressed comments from QC.
0.6	29/11/2016	Cristiano Giuffrida	Preliminary release for the QC.
0.5	29/11/2016	Marinos Tsantekidis	Integrated TUBS' contribution.
0.4	28/11/2016	Elias Athanasopoulos	Integrated FORTH's contribution.
0.3	27/11/2016	Dmitry Pidan	Integrated IBM's contribution.
0.2	24/11/2016	Cristiano Giuffrida	Integrated feedback from all partners.
0.1	16/11/2016	Cristiano Giuffrida	Document structure.

Contents

1 Introduction 9

2 Background 11
2.1 Attack primitives . . . 11
2.2 Implementation constraints . . . 13
2.3 Mapping applications . . . 14
2.3.1 Implantable medical device . . . 14
2.3.2 Automotive . . . 15
2.3.3 Cloud . . . 15

3 Overview 17
3.1 Memory error vulnerabilities . . . 17
3.1.1 Buffer overrun and underrun . . . 18
3.1.2 NULL pointer dereference . . . 19
3.1.3 Integer overflow . . . 20
3.1.4 Use-after-free . . . 20
3.1.5 Uninitialized read . . . 21
3.1.6 Type confusion . . . 22
3.2 Design overview . . . 23
3.2.1 Software verification . . . 23
3.2.2 Software testing . . . 24
3.2.3 Memory error defenses . . . 24
3.2.4 Exploitation defenses . . . 25
3.2.5 Integrity monitoring . . . 25
3.2.6 Recovery . . . 25
3.3 Residual attack surface . . . 26

4 **Design** 27
- 4.1 Software verification . . . 27
  - 4.1.1 ExpliSAT . . . 28
- 4.2 Software testing . . . 32
  - 4.2.1 VUzzer . . . 32
- 4.3 Memory error defenses . . . 34
  - 4.3.1 Memory error detection: TypeSan . . . 34
  - 4.3.2 Memory error containment: VTPin . . . 36
- 4.4 Exploitation defenses . . . 37
  - 4.4.1 Passive defenses: Anti-ROP . . . 38
  - 4.4.2 Active defenses . . . 40
  - 4.4.3 GPGPU network intrusion detection systems . . . 46
- 4.5 Integrity monitoring . . . 47
  - 4.5.1 GRIM . . . 47
- 4.6 Recovery . . . 50
  - 4.6.1 OSIRIS . . . 50
- 4.7 Web and mobile security and privacy . . . 52
  - 4.7.1 CCSP . . . 52
  - 4.7.2 Constant blinding in JIT engines . . . 53
  - 4.7.3 Proxy injection detection . . . 54
  - 4.7.4 Sandboxing mobile application on the cloud . . . 55
  - 4.7.5 Anti-track droid . . . 56
  - 4.7.6 REAPER . . . 57

5 **Conclusion** 59

List of Figures

2.1 Sample attack primitive implementations 12
2.2 The SHARCS models and implementation constraints 13
2.3 The Neurasmus-SoC which implements the implant application 14
2.4 The AUTOSAR software stack 15
2.5 End-to-end view of OnApp platform 16

3.1 Evolution of memory error and web vulnerabilities/exploits . 18
3.2 Design overview of the SHARCS software technologies 24

4.1 C code snippet demonstrating path explosion problem 29
4.2 CFG of the code demonstrating path explosion problem 30
4.3 Anti-ROP Architecture 39
4.4 The GPGPU NIDS architecture 47

1 Introduction

Work-package 4 (WP4) is the core package dedicated to the design and implementation of the SHARCS software technologies. In particular, WP4 pursues two important goals. The first goal is to offer support for hardware technologies devised by WP3 that need special modifications in the software layer. The second and more important goal of WP4 is to devise software-based solutions where hardware security techniques are not applicable or available. The first goal concentrates on software support facilities which have been briefly described elsewhere (D4.1) and tailored to hardware techniques detailed in D3.2. For this reason, we focus D4.2 on the second goal, which serves as a basis for the SHARCS core software technologies.

In detail, this deliverable reports on our efforts in T4.2, T4.3, and T4.4 to support security by design at the software layer. Our security philosophy dictates a holistic approach to systems security across several dimensions. First, it requires WP4 to devise *comprehensive* security technologies that cover all the core requirements identified by the analysis performed in D2.1. This is crucial to support end-to-end security. Second, it requires WP4 to devise *versatile* security technologies that cover all the application domains and constraints required by WP5. Third, it requires WP4 to devise *generic* security technologies that can shield all the layers of the software stack (hypervisor, operating system, applications, etc.) from attacks. Finally, it requires WP4 to devise *multilevel* security technologies that can provide defenses in depth at different stages of the software lifecycle.

We start by consolidating the application requirements in Chapter 2. The goal is to map our requirements into attack primitives that make part of our threat model, and implementation constraints, in the different application domains considered in WP5. Identifying the implementation constraints is important to ensure our design can effectively comply with the restrictions, but also benefit from the opportunities offered by the different application domains in WP5. Identifying the attack primitives is important to ensure

that our design can counter all the attacks and threats considered in our requirements analysis.

Before presenting a design overview of our software technologies, Chapter 3 further analyzes the security vulnerabilities that allow adversaries to craft such attack primitives. We note that our analysis also provides design guidelines for the technologies developed by WP3. We report on the attack primitives and vulnerabilities in D4.2 rather than D3.2, given that our threat model focuses on software rather than hardware vulnerabilities. We conclude by introducing our design and detailing several representative software technologies at different design layers in Chapter 4. We also show that our design covers all the dimensions of our security philosophy.

2 Background

This chapter summarizes the key properties of the applications in WP5 and maps our requirements analysis into concrete design guidelines (rather than technical software requirements, as detailed in D4.1) for our software technologies. We first outline the general classes of attack primitives and implementation constraints that are relevant for our requirements. Next, we briefly recap the properties of each of the applications in WP5 and map their requirements to the attack primitives and implementation constraints previously outlined. The following analysis is essential to cover two dimensions of our security philosophy: *comprehensiveness* and *versatility*. The other dimensions are covered by our system design in the following chapters.

2.1 Attack primitives

According to our threat model defined in D2.1, SHARCS technologies need to generally secure systems across all the dimensions dictated by the "CIA" model: *confidentiality*, *integrity*, *availability*[1]. To breach confidentiality, an attacker needs to craft primitives to leak security-sensitive data from memory, thus operating *unauthorized memory reads*. To breach integrity, an attacker needs to craft primitives to tamper with security-sensitive data in memory, thus operating *unauthorized memory writes*. As detailed in Chapter 3, such primitives are easy to craft in practice due to widespread memory error vulnerabilities in systems software. These two primitives alone can be combined to craft more complex primitives and mount end-to-end attacks.

For example, tampering with data that affects the control flow of a target application (i.e., *control data*) allows an attacker to craft an *unauthorized execution* primitive. This primitive can be used to divert normal control flow and grant an attacker arbitrary code execution capabilities. These can, for

[1]The other dimensions of the generalized "CIANA" model in D2.1 are addressed by WP3

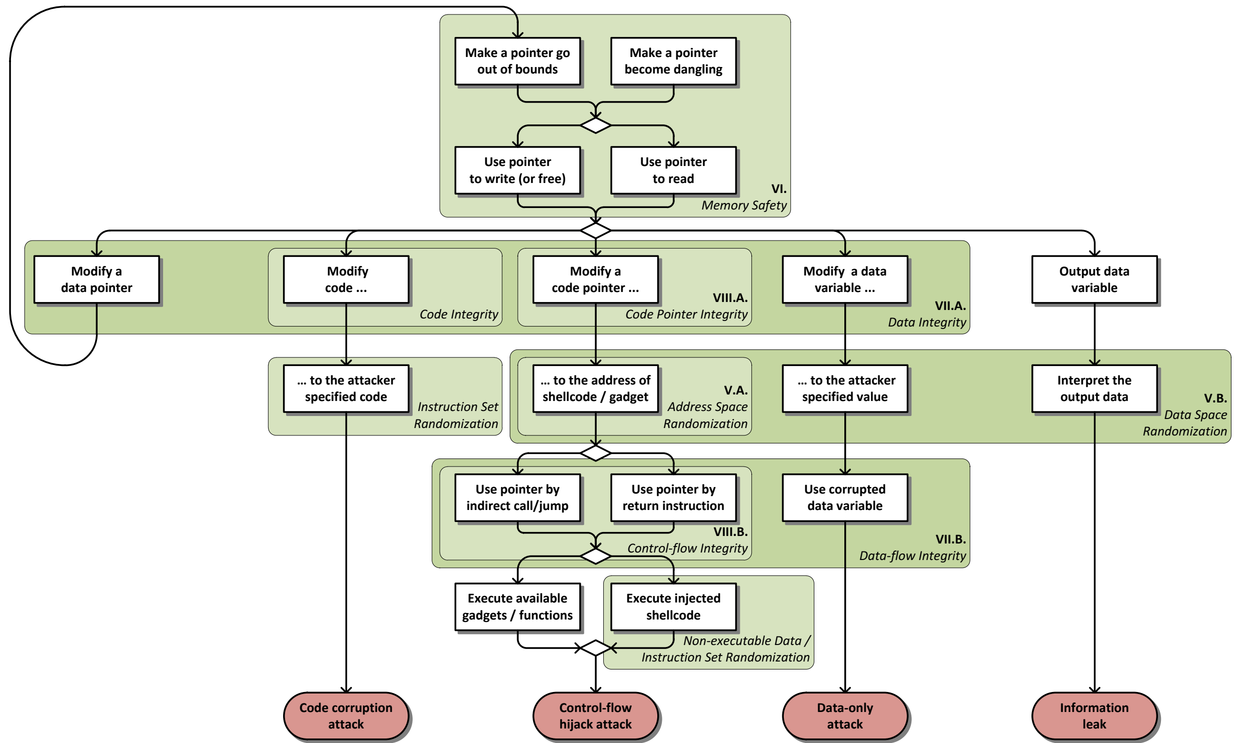

Figure 2.1: Sample attack primitive implementations and their application to mount different attack types (Source: Szekeres et al. [85])

example, be used to stop execution (breaching availability) but, more importantly, to leak or tamper with security-sensitive data in memory (breaching confidentiality or integrity, respectively). In other words, an unauthorized execution primitive poses the highest security threat.

We note that these three fundamental primitives can be composed in different ways, by an attacker, to mount arbitrarily sophisticated attacks. Figure 2.1 illustrates a number of examples. As shown in the figure, there are several options to implement the attack primitives considered, on top of existing memory error vulnerabilities (see Chapter 3), and use them in different attack types. For example, other than relying on a *control-flow diversion* attack, an attacker can opt, when possible, for a *code corruption attack* to execute arbitrary code. In a code corruption attack, an attacker uses an unauthorized memory write primitive to corrupt executable code memory and inject their own malicious code. In an *information leak attack*, an attacker uses an unauthorized memory read primitive to leak control data (e.g., a return address on the stack) or non-control data (e.g., a user ID on the heap) from the application. In a *data-only attack*, finally, an attacker uses an unauthorized memory write primitive to corrupt non-control data and gain increased privileges (e.g., the administrator user ID).

2.2 Implementation constraints

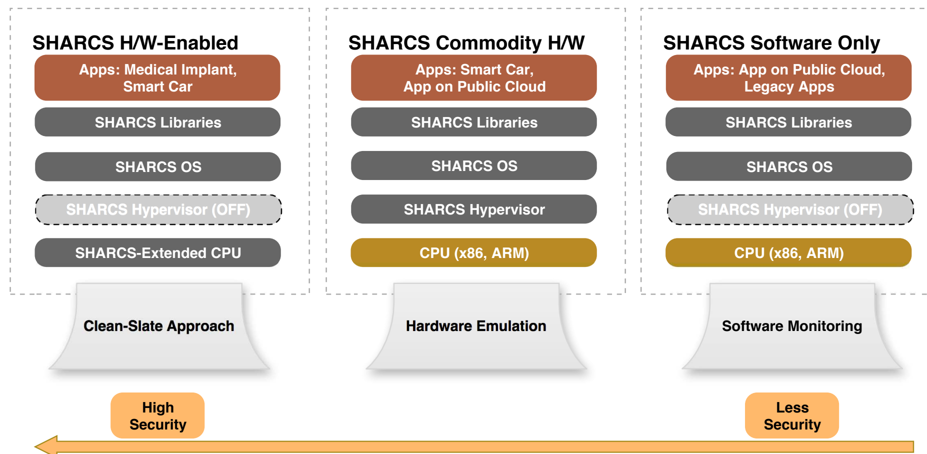

Figure 2.2: The SHARCS models and implementation constraints

Figure 2.2 recaps the canonical SHARCS models, which cover all the application domains in WP5 and immediately suggest their implementation constraints. The *clean-slate approach* model allows unrestricted hardware modifications and is the focus of WP3. The *hardware emulation* and the *software monitoring* models are of core interest for WP4.

The hardware emulation model assumes a SHARCS-hardened software stack running on top of commodity hardware. While this model prevents any underlying hardware modifications, it allows software technologies to leverage commodity hardware features to improve security and also emulate custom hardware features developed by WP3 (e.g., by using hardware virtualization). In addition, depending on the particular setting, our technologies can be used to protect source or binary programs (i.e., legacy executable binaries whose source code is not available). In other words, this model yields two classes of software technologies: (i) *commodity hardware-supported, source-level*; (ii) *commodity hardware-supported, binary-level*.

The software monitoring model assumes a SHARCS-hardened software stack running on arbitrary hardware platforms. As a result, this model constrains our software technologies to make no assumptions on the underlying hardware or software platform. This prevents our technologies from operating hardware modifications, leveraging commodity hardware features, or introducing any hypervisor-based techniques. It only allows for security features to be implemented entirely in software, at the source or binary level. In other words, we can implement two other classes of software technologies in this model: (i) *software-only, source-level*; (ii) *software-only, binary-level*. The emphasis on binary-level techniques is particularly prominent, given

that it becomes more common for software technologies to be deployed on legacy applications where there is no source code available.

2.3 Mapping applications

In this section, we briefly recap the properties of the applications in WP5 and evidence the relevant attack primitives and implementation constraints for each application.

2.3.1 Implantable medical device

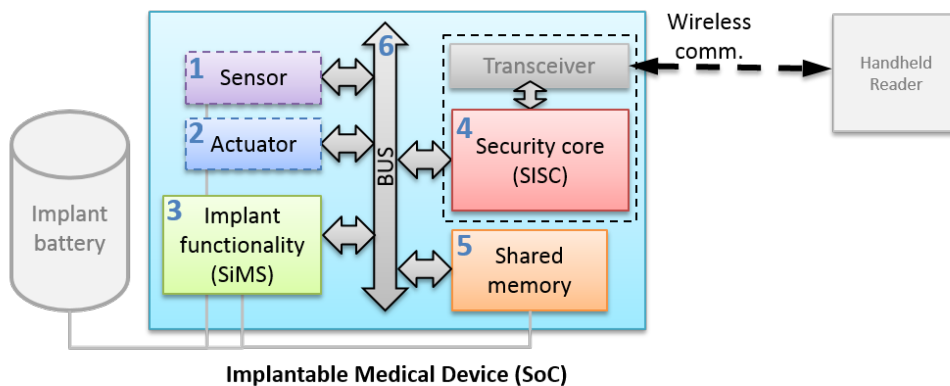

Figure 2.3: The Neurasmus-SoC which implements the implant application

The implantable medical device application (Figure 2.3) is an instance of the *clean-slate approach* SHARCS model, allowing for security technologies to be implemented in hardware, in software at the source level, and in software at the binary level, with no restrictions.

We now briefly revisit the application requirements from D2.1 to understand the impact of the attack primitives introduced earlier on the implantable medical device application. SR-1.1 and SR-1.4 only deal with accessibility and extra-functional requirements for defenses and are not relevant for the discussion here. SR-1.2 dictates immutability of treatment delivery, which can be compromised by an attacker armed with unauthorized execution or memory write primitives (e.g., by changing treatment parameters). SR-1.4 dictates confidentiality of patient data, which can be compromised by an attacker armed with unauthorized execution or memory read primitives (e.g., by leaking patient data over the network). SR-1.5 dictates that certain (maintenance) operations should only be granted to privileged users, which can be compromised by an attacker with unauthorized execution or memory write primitives (e.g., by evading privilege checks).

In short, countering all the attack primitives, the unauthorized execution primitive above all, is necessary to satisfy our security requirements.

2.3.2 Automotive

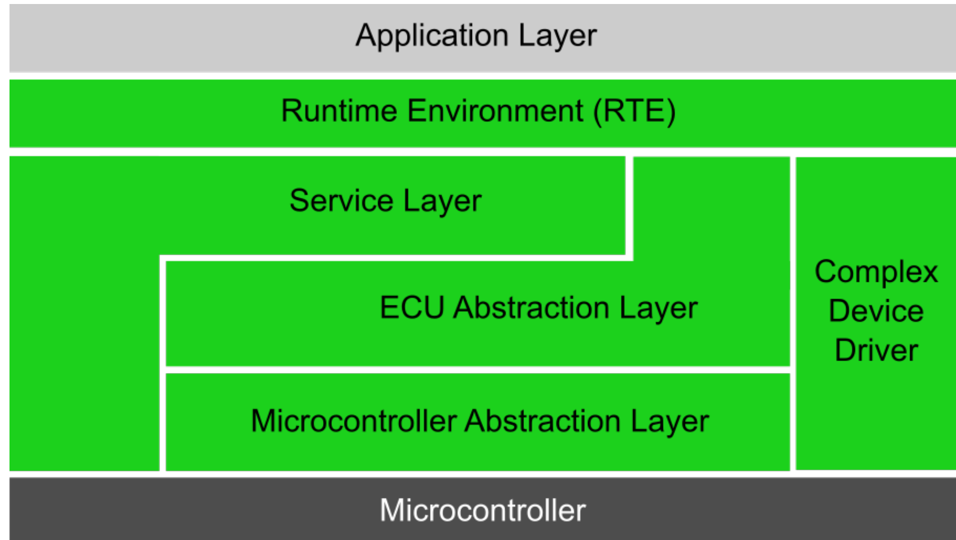

Figure 2.4: The AUTOSAR software stack

The automotive application (Figure 2.4) can be considered an instance of the *clean-slate approach* or of the *hardware emulation* SHARCS model depending on the particular setting. This generally allows for security technologies to be implemented in hardware, or in software (source-level and binary-level) supported by commodity hardware features.

We now briefly revisit the application requirements from D2.1 to understand the impact of the attack primitives introduced earlier on the automotive application. SR-2.1, SR-2.2, and SR-2.3 dictate no arbitrary code execution via message, data flash, or controller manipulation, which can be compromised by an attacker armed with unauthorized execution primitives (e.g., by executing code to control critical ECUs). SR-2.4 dictates no cross-controller information leak, which can be compromised by an attacker with unauthorized execution or memory read primitives (e.g., by reading private user data from another controller). In short, countering unauthorized execution primitives above all, and the other primitives as a by-product or to prevent information leaks, is necessary to satisfy our security requirements.

2.3.3 Cloud

The cloud application (Figure 2.5) can be considered an instance of the *hardware emulation* or of the *software monitoring* SHARCS model depending

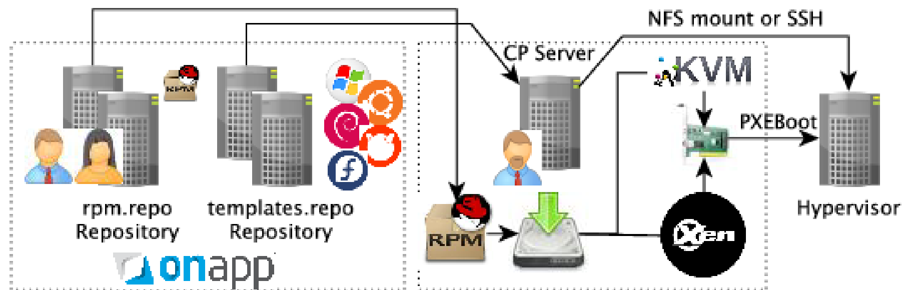

Figure 2.5: End-to-end view of the OnApp platform from the repository through to deployment

on the particular setting. This allows for security technologies to be implemented entirely in software or in software leveraging features of commodity hardware. Both source-level and binary-level solutions are possible, but the emphasis on binary-level technologies is important given their deployability and backward compatibility benefits in a cloud (virtualization) setting.

We now briefly revisit the application requirements from D2.1 to understand the impact of the attack primitives introduced earlier on the cloud application. SR-3.1 dictates confidentiality of VM data, which can be compromised by an attacker armed with unauthorized execution or memory read primitives (e.g., by reading the cryptographic keys of a VM user). SR-3.2 dictates integrity of VM data, which can be compromised by an attacker armed with unauthorized execution or memory write primitives (e.g., by tampering with the workload of a VM user). SR-3.3 dictates availability of the platform, which can be compromised by an attacker armed with any unauthorized primitive causing disruption of service (e.g., by operating an invalid memory access and bringing the system to a halt). SR-3.3 and SR-3.4 allow only authenticated operations attributable to a specific user, which can be compromised by an attacker armed with unauthorized execution primitives (e.g., performing an administrative operation as a regular user). In short, countering all the attack primitives, and, again, the unauthorized execution primitive above all, is necessary to satisfy our security requirements.

3 Overview

This chapter provides a high-level overview of the SHARCS software technologies, based on the analysis detailed in Chapter 2. For this purpose, we first provide a comprehensive analysis of all the relevant memory error vulnerabilities enabling an attacker to craft the attack primitives detailed in the previous chapter. Next, we illustrate how our high-level design can provide several layers of generic and in-depth software defenses, to counter such vulnerabilities and the corresponding attacks. The following analysis is essential to cover the two remaining dimensions of our security philosophy: *generic* and *multilevel* applicability. We conclude with a brief discussion on the residual attack surface.

3.1 Memory error vulnerabilities

Memory error vulnerabilities constantly rank as the first security threat for systems software written in low-level languages such as C and C++. Such languages give programmers full control over memory and make no effort to enforce type and memory safety. As a result, programming mistakes can easily lead to bugs that induce the program to use memory in unintended ways. Such memory error bugs are often the cause of security vulnerabilities in the common case where an attacker can successfully trigger and exploit the underlying bugs. This allows an attacker to craft memory read, write, and execute primitives and gain increased privileges.

Figure 3.1 depicts the evolution of memory error vulnerabilities and exploits over time. This plot has been generated using an open-source analysis tool [6] with vulnerability data from the CVE data feed [3] and exploit data from the Exploit Database [5]. As shown in the figure, memory error vulnerabilities are a very prominent class of vulnerabilities and, in recent years, compete in number and exploit coverage with web vulnerabilities—the other major class of security vulnerabilities. Their evident real-world

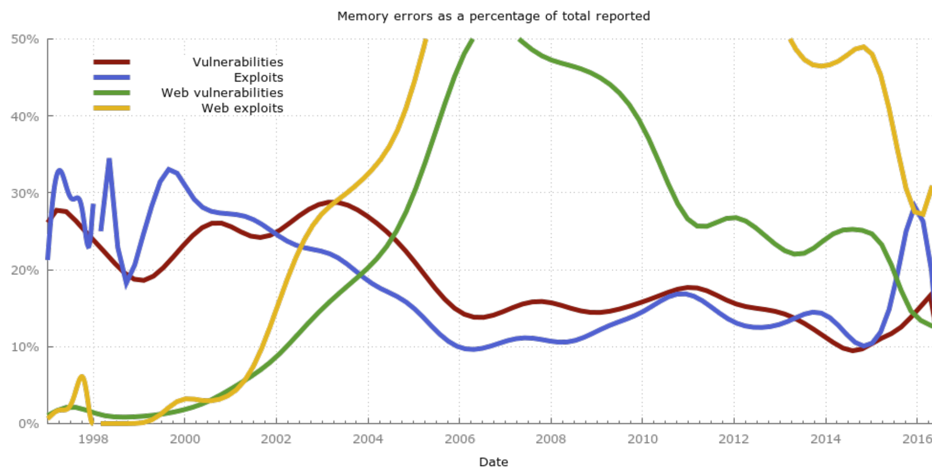

Figure 3.1: Evolution of memory error and web vulnerabilities/exploits

impact, along with their prevalence in systems software, justifies our focus on memory error vulnerabilities in this project.

In the following sections, we will briefly introduce all the major classes of memory errors, only excluding those that have limited security implications or are already countered by practical and widespread security hardening techniques (e.g., format string errors).

3.1.1 Buffer overrun and underrun

```
1  char *lccopy(const char *str) {
2    char buf[BUFSIZE];
3    char *p;
4
5    strcpy(buf, str);
6    for (p = buf; *p; p++) {
7      if (isupper(*p))
8        *p = tolower(*p);
9    }
10   return strdup(buf);
11 }
```

Listing 3.1: A sample function containing a buffer overrun vulnerability (Source: https://www.owasp.org).

A buffer overrun (or underrun) occurs when a program reads or writes memory outside the intended buffer (or array) boundaries. Such unintended out-of-bounds accesses are typically termed buffer overflows (or underflows) when the program writes data after the end (or before the beginning) of the buffer and buffer overreads (or underreads) when the program reads data after the end (or before the beginning) of the buffer. In both

cases, this *spatial memory error* leads to undefined behavior, as the outcome depends on the data stored in adjacent memory locations. This vulnerability can be exploited by an attacker to read/write (or indirectly execute) from/to unintended locations in memory.

Listing 3.1 shows an example of a typical buffer overrun vulnerability. The function **lccopy** in the example receives a string input and returns a lower-case copy of the string in output. The C function expects the input string to be always smaller than **BUFSIZE**. However, if an attacker can lure the program into using a larger, attacker-controlled string, the **strcpy** invocation at line 5 will overrun the **buf** buffer's boundary and corrupt adjacent data. This may, for example, allow an attacker to overwrite the return address on the stack and execute arbitrary code upon function return.

3.1.2 NULL pointer dereference

```
1 void png_copy(png_structp png_ptr, int length, const void
2 *user_data) { png_charp chunkdata;
3   chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
4   /* ... */
5   memcpy(chunkdata, user_data, length);
6   /* ... */
7  }
```

Listing 3.2: A sample function containing a NULL pointer dereference vulnerability (Source: https://www.securecoding.cert.org).

A NULL pointer dereference occurs when a program uses a NULL pointer to read/write from/to memory. When the NULL pointer is used to access memory directly, such unintended access can be normally detected by the hardware (e.g., by ensuring the zero page is always unmmaped in a virtual memory address space organization). More problematic are cases when a pointer is derived from a NULL pointer through pointer arithmetic and later used to access memory. In this case, the program may use the derived pointer to read/write from/to higher addresses in memory, leading to undefined behavior. This vulnerability can be exploited by an attacker to read/write (or indirectly execute) from/to unintended locations in memory.

Listing 3.2 shows an example of a typical NULL pointer dereference vulnerability. The function **png_copy** in the example is derived from a real-world code snippet in a vulnerable version of the libpng library. The C function dynamically allocates a buffer of size **length+1** pointed by **chunkdata** and copies **user_data** into it. However, if an attacker can lure the program into using a **length** with the value **-1**, the **chunkdata** pointer will be set to NULL and the **memcpy** invocation at line 5 will trigger a NULL pointer dereference. This may, for example, allow an attacker to overwrite security-sensitive data structures stored in the low part of the memory address space.

3.1.3 Integer overflow

```
1 cairo_status_t _cairo_pen_init (...) {
2   /* ... */
3   pen->num_vertices = _cairo_pen_vertices_needed(
4     gstate->tolerance, radius, &gstate->ctm
5   );
6   pen->vertices = malloc(
7     pen->num_vertices * sizeof(cairo_pen_vertex_t) );
8   /* ... */
9 }
```

Listing 3.3: A sample function containing an integer overflow vulnerability (Source: https://www.securecoding.cert.org).

An integer overflow occurs when the program performs arithmetic operations that exceed the maximum size of the integer type used to store the result. For example, `LONG_MAX+1` causes an integer overflow for a `long` data type. Integer overflows are sometimes benign arithmetic errors explicitly handled by C/C++ programs. When integer overflows are accidental, however, they can lead to undefined behavior and often to memory errors. For example, if an integer affected by overflow is used as an index to a buffer, the corresponding memory access may result in a buffer overrun (or underrun) and allow an attacker to read/write (or indirectly execute) from/to unintended locations in memory.

Listing 3.3 shows an example of a typical integer overflow vulnerability. The function `_cairo_pen_init` in the example is derived from a real-world integer overflow vulnerability in the Mozilla Scalable Vector Graphics (SVG) viewer. The C function allocates a `pen->vertices` buffer whose size is based on the multiplication of a signed integer and an unsigned integer. If the resulting value is too large to be stored in a `size_t` data type, this will cause an unsigned integer wrap at line 7 and result in allocating memory of insufficient size. This may cause later accesses to the buffer to read/write memory past the intended boundary, allowing an attacker to exploit the resulting buffer overrun and access adjacent data in memory.

3.1.4 Use-after-free

A use-after-free occurs when the program uses a pointer to a deallocated object (i.e., a *dangling pointer*) to read/write from/to memory. This *temporal memory error* leads to undefined behavior, as the outcome depends on the data stored in the underlying memory location after deallocation. For example, if the underlying memory is reused to store a new object, a write-after-free operation can corrupt the data in the new object. This vulnerability can be exploited by an attacker to read/write (or indirectly execute) from/to unintended locations in memory.

```
1  void run_op(void) {
2    char* ptr = (char*)malloc (SIZE);
3    /* ... */
4    if (err) {
5      abrt = 1;
6      free(ptr);
7    }
8    /* ... */
9    if (abrt) {
10     logError("operation aborted before commit", ptr);
11   }
12 }
```

Listing 3.4: A sample function containing a use-after-free vulnerability (Source: https://www.owasp.org).

Listing 3.4 shows an example of a typical use-after-free vulnerability. The C function **run_op** in the example first allocates a buffer of size **SIZE** pointed by the pointer **ptr**. In case of error, the buffer is deallocated and the corresponding pointer becomes dangling. The **logError** invocation at line 10 will then trigger a use-after-free and read from previously deallocated memory. This may, for example, allow an attacker to leak security-sensitive data from a newly allocated object reusing the underlying memory location.

3.1.5 Uninitialized read

```
1  void report_error(const char *msg) {
2    const char *error_log;
3    char buffer[BUFFERSIZE];
4
5    sprintf(buffer, "Error: %s", error_log);
6    printf("%s\n", buffer);
7  }
```

Listing 3.5: A sample function containing an uninitialized read vulnerability (Source: https://www.securecoding.cert.org).

An uninitialized read occurs when the program reads uninitialized data from a newly allocated object. This *temporal memory error* leads to undefined behavior, as the outcome depends on the data stored in the underlying memory before the allocation. This vulnerability can be exploited by an attacker to read (or indirectly execute) from unintended locations in memory.

Listing 3.5 shows an example of a typical uninitialized read vulnerability. The C function **report_error** in the example formats and writes an error string to standard output. However, the function fails to initialize the **error_log** local variable, causing the **sprintf** invocation at line 5 to trigger an uninitialized read. This may, for example, allow an attacker controlling the value of the uninitialized pointer on the stack to leak security-sensitive data or trigger other vulnerabilities. For example, if the attacker lures the

program into pointing **error_log** to a string with more than **BUFFERSIZE** characters, the **sprintf** invocation will result in a buffer overflow.

3.1.6 Type confusion

```
1  class IShouldRunCalculator { public: virtual bool UWannaRun() = 0; };
2
3  class CalculatorDecider final : public IShouldRunCalculator {
4  public:
5    CalculatorDecider() : m_run(false) {}
6    virtual bool UWannaRun() { return m_run; }
7  private: bool m_run;
8  };
9
10 class DelegatingCalculatorDecider final : public IShouldRunCalculator {
11 public:
12   DelegatingCalculatorDecider(IShouldRunCalculator* d) : m_delegate(d) {}
13   virtual bool UWannaRun() { return m_delegate->UWannaRun(); }
14 private: IShouldRunCalculator* m_delegate;
15 };
16
17 int calculate() {
18   CalculatorDecider nonono;
19   DelegatingCalculatorDecider isaidno(&nonono);
20   IShouldRunCalculator* decider = &isaidno;
21   CalculatorDecider* confused =
22     reinterpret_cast<CalculatorDecider*>(decider);
23   if (confused->UWannaRun())
24   execl("/bin/gnome-calculator", 0); }
```

Listing 3.6: A sample function containing a type-confusion vulnerability (Source: https://googleprojectzero.blogspot.nl).

A type confusion bug occurs when the program is *"confused"* about the type of a given object and mistakenly uses a pointer of type T_1 to read/write from/to an object of type T_2. This type-unsafe access leads to undefined behavior, as the outcome depends on the in-memory structure of type T_1 vs. type T_2 and the data of type T_2 stored in the underlying memory location. This memory error is particularly common in C++ programs, which often use fast (but error-prone) C-style casts to address different objects within a complex class hierarchy. The net effect is similar to a use-after-free, where the underlying memory is reused by a newly allocated object of a different type. This vulnerability can be exploited by an attacker to read/write (or indirectly execute) from/to unintended locations in memory.

Listing 3.6 shows an example of a typical type confusion vulnerability. The C++ function **calculate** first casts a pointer into an object of type **IShouldRunCalculator** to a pointer of type **CalculatorDecider***. Later (line 23), it dereferences the casted pointer to invoke the **UWannaRun** method and decide whether to run the calculator. In the example, the check will take place using code and data from an object of an incompatible type. This

may, for example, allow an attacker, controlling the target object, to execute arbitrary code upon the invocation of the `UWannaRun` method.

3.2 Design overview

Having reviewed the major classes of memory error vulnerabilities and exemplified the corresponding exploits part of our threat model, we now present a design overview of the SHARCS software technologies developed by WP4. Our framework follows a *multilevel* approach to counter memory error vulnerabilities and the corresponding attacks, deploying a portfolio of security-enhancing techniques at different stages of the software life-cycle. At each stage, we develop *generic* and *versatile* techniques, that can be used at different layers in the software stack (OS, application, etc.) and in the different application domains with their own implementation constraints (binary-only, hardware-supported, etc.). This includes a variety of software techniques depending on the particular layer considered, including program analysis tools, compilers, binary translators, run-time systems, etc. The design is intentionally modular and generic, so that we can easily integrate the different techniques in an optimal way in the different application domains.

Figure 3.2 outlines our approach. We represent a piece of software as a blackbox, originally containing an unspecified number of memory error bugs. Our goal is to eliminate such bugs or prevent them from being the source of successfully exploitable vulnerabilities after deployment in production. For this purpose, T4.2 and T4.4 develop technologies that incrementally reduce the attack surface at each stage of the software life-cycle (with the exception of software verification, developed by T4.3). This is to minimize the exploitation power left to an attacker. The following subsections present a detailed overview of our technologies at each design layer.

3.2.1 Software verification

Our first step (①) towards securing software against memory error vulnerabilities is to verify software components for correctness. For this purpose, we develop software verification techniques that can provably assert that a target software component is free of bugs, reducing the attack surface. If the verification fails, our solution provides detailed reporting on the vulnerabilities found (T4.3). For verification to work, we need a formal specification or a model of the software being developed. While this strategy is effective at proving software for correctness, it has trouble scaling to complex software components. For this reason, we limit the scope of our verification techniques to simple software modules.

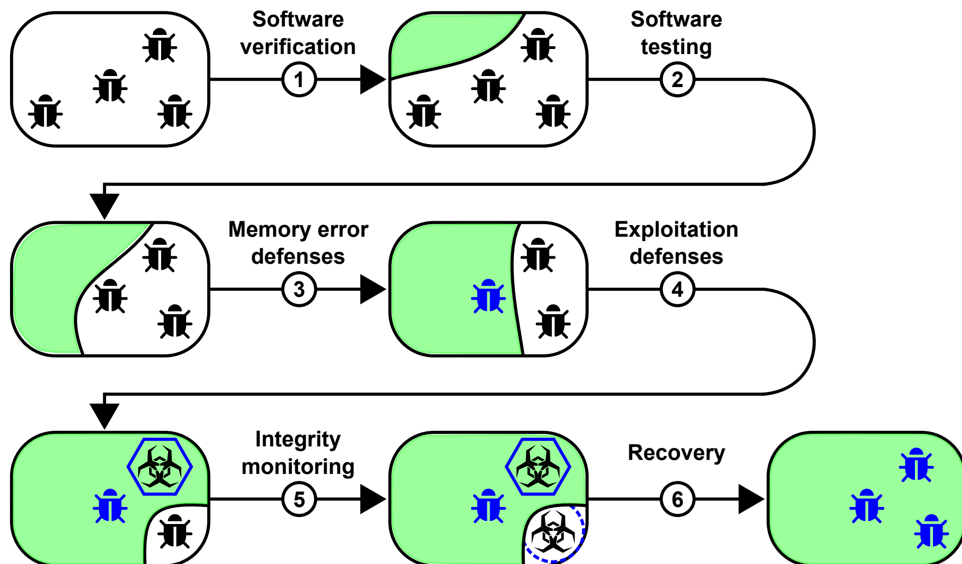

Figure 3.2: Design overview of the SHARCS software technologies and their application at different stages of the software lifecycle. The green areas mark the parts of the software secured by our technologies. The blue marks highlight the impact of our hardening techniques

3.2.2 Software testing

Complex software components, that are not suitable candidates for verification techniques, can still contain exploitable bugs. Our second step (②) is to attempt to eliminate such bugs, by developing software testing techniques, and further reduce the attack surface. Such techniques run the target software through a variety of carefully crafted inputs in an attempt to maximize execution coverage and find bugs. Once bugs are found, they can be fixed during development before they make their way to production. While high-coverage testing is effective at eliminating some bugs, it cannot generally eliminate all the bugs. Residual bugs, that are only triggered under special conditions, escape testing and still plague running software in production.

3.2.3 Memory error defenses

Once residual bugs make their way to production, they may allow an attacker to exploit the corresponding memory error vulnerabilities in running software. To further reduce the attack surface, our third step (③) is to counter any attempt to exploit such vulnerabilities by developing memory error defenses. Such defenses can prevent or detect an exploitation attempt for each class of memory error vulnerabilities (e.g., buffer overflow, use-after-free, etc.) and stop exploits from the get-go. While such techniques

are effective at countering attacks, they cannot generally prevent or detect all the exploitation attempts due to inherent limitations (e.g., custom memory allocators). In addition, deploying many memory error defenses to cover all the classes of vulnerabilities may easily exhaust the performance budget available in some application domains; hence, other solutions are needed.

3.2.4 Exploitation defenses

Exploitation attempts that escape our memory error defenses may still allow an attacker to mount successful attacks. To further reduce the attack surface, our fourth step (④) is to prevent any exploitation attempt from successfully reaching its target by developing exploitation defenses. Such defenses ignore the underlying cause (i.e., memory error) of each exploit and focus on efficiently preventing or detecting successful exploitation using a variety of solutions, including randomization, isolation, and policy enforcement techniques (as detailed in Section 4.4).

3.2.5 Integrity monitoring

While our defenses are effective at efficiently stopping most real-world attacks, one cannot rule out implementation bugs in the defenses themselves, or sophisticated attacks that bypass our multiple lines of defense, let alone attacks that are not even part of our threat model. For this reason, our fifth step (⑤) is to develop integrity monitoring techniques that can constantly check the integrity of running software, to detect successful exploitation attempts. Once any such attempt is detected, recovery measures can be taken to resume execution in a sane state.

3.2.6 Recovery

Our sixth and final step (⑥) is to develop error recovery techniques that can revert the execution of running software to a safe state, once erroneous conditions are detected. Our techniques rely on efficient memory checkpointing to periodically save the state of the running software, and support switching to a saved checkpoint on demand. Recovery techniques are useful to resume the execution in a benign state after detecting a memory error, an exploitation attempt, or a software integrity violation, using our core software technologies. In other words, our core software technologies are useful to virtualize a memory error exploit into a generic "fail-stop error", while our recovery techniques are useful to virtualize the latter into a "recoverable software error". This strategy allows us to resume execution in a consistent pre-exploitation state. In practice, recovering failed software components poses nontrivial challenges and may require intervention in both the software and the runtime. These issues are better detailed in Section 4.6.

3.3 Residual attack surface

While SHARCS embraces an end-to-end approach to systems security in the setting considered, it is important to challenge some of our own assumptions and analyze the residual attack surface left to an attacker. For this reason, as part of WP4, we also consider sophisticated attack techniques that can bypass even advanced defenses and carefully evaluate their impact. We are specifically interested in evaluating the strength of state-of-the-art (and our own) software defenses against attacks not included in our threat model, with a particular focus on side-channel attacks and hardware fault attacks. This is work in progress and we will further report on our efforts in D4.3.

4 Design

This chapter details the design of the SHARCS software technologies. For each design layer, we outline the contributions made by SHARCS software technologies and illustrate the design of one or more leading solutions in different application domains. This is meant to exemplify the solutions developed by T4.2, T4.3, and T4.4 in every given design layer, but without aiming for completeness. For a more complete and up-to-date view of the solutions developed in every design layer, we refer the reader to the SHARCS publication page[1].

Furthermore, we use Control-Flow Integrity (CFI) exploitation defenses as pilot technologies to illustrate the versatility dimension of our security philosophy. For this purpose, we present different CFI solutions that automatically balance performance and security in an optimal way, based on the given implementation constraints. We focus on CFI, given that its goal is to enforce statically derived control-flow invariants during the execution, to detect control-flow diversion attacks. This thwarts unauthorized execution primitives, which we identified as the most general and damaging primitives based on our requirements (Chapter 2).

4.1 Software verification

To verify software components for correctness, T4.3 develops scalable vulnerability reporting tools based on symbolic interpretation. These tools are applicable when the source code of a given component is available. In the following, we present the design of the ExpliSAT tool as the leading software verification technology developed in WP4.

[1] http://www.sharcs-project.eu/publications

4.1.1 ExpliSAT

The notion of *concolic verification* is typically used in the context of static analysis and dynamic testing and refers to a hybrid software verification technique that combines concrete values of control variables with symbolic representation of data. This technique is now used in many *symbolic execution* tools, such as *DART* [37], *CUTE* [75], *KLEE* [20], and others. The tool we use in the framework of the SHARCS project, *Explisat* [17], is based on a similar algorithm, but instead of executing the program on each of the control paths, it, essentially, builds a model of each explicit control flow path with symbolic data variables and invokes a decision procedure (*SAT*) to determine whether there exists an instantiation of this path that violates correctness assertions. This technique is called *symbolic interpretation*, since rather of executing each control path, the tool actually interprets every instruction symbolically, getting at the end a symbolic formula that represents a set of all feasible input values for this execution path. By solving this symbolic formula, we can find a set of concrete input values for this path, thus if the path leads to an error we end up with the set of input values required for realization of this error.

Symbolic interpretation is an extremely powerful tool when looking for the corner case behaviors of the application under test. Since we use symbolic representation for each input, the algorithm is completely independent of having any prior knowledge about "interesting scenarios" that usually comprise a set of testing data, but rather discovers those scenarios itself. This way, it becomes an ultimate method for finding unknown behaviors in the application, part of which can actually be the security vulnerabilities, exploitable by some corner case, carefully crafted input.

To detect erroneous scenarios, ExpliSAT relies on user-provided functional checks (a.k.a assertions), as well as a reach set of built-in checks, that cover most of the memory error vulnerabilities (see Section 3.1). Those checks are performed during the interpretation without a need to specify them as assertions. Below is a partial list of those checks:

- **Valid pointer dereferencing** – on every pointer dereferencing operation, ExpliSAT checks that the pointer is properly assigned, i.e. points to a valid memory block.
- **Buffer overflow** – on every memory access, ExpliSAT verifies that the access is within the allocated chunk.
- **Variable initialization** – on every variable access, ExpliSAT checks that the variable was assigned previously.
- **Memory leaks** – at the end of every execution path, ExpliSAT checks that all memory chunks that were dynamically allocated are released (this check is enabled via command-line option).

```
1  void foo()
2  {
3    int x;
4    int result = 0;
5    int counter = 0;
6
7    while (counter < 50) {
8      x = nondet_int();
9      if (x > 0) {
10       result = result + 1;
11       if (result % 2 == 1) {
12         result = result * 2;
13       }
14     }
15     else {
16       result = result - 1;
17     }
18     counter = counter + 1;
19   }
20 }
```

Figure 4.1: C code snippet demonstrating path explosion problem

Contributions Symbolic interpretation explicitly traverses each and every feasible execution path in the application. This is an inherent weakness of the method – in the real-scale applications, number of such paths grows exponentially, making it impossible to cover the whole input space. Moreover, the tool may become "stuck" in some corner case behavior, exploring more and more newly created paths, and miss errors in other directions. In the next section, we present a novel algorithm that aims at coping with this problem, by dynamically reducing the number of explicitly explored paths using path merging[2].

Branch reduction algorithm Symbolic interpretation is an efficient technology for data handling, being able to process a bunch of values of variables at every step. On the other hand, program control is processed explicitly, s.t. the engine traverses every path in the Control Flow Graph (*CFG*). Inherently, this technique introduces a drawback, known as the *path explosion problem*. When a series of n multiple if-statements is present in a program, the number of possible execution paths may grow exponentially up to 2^n paths, thus making a symbolic execution engine run virtually forever. To illustrate the problem and the ExpliSAT solution, we use the following running example. Consider the simple C code snippet, shown in Figure 4.1, and its corresponding CFG, presented in Figure 4.2.

[2]Patent 15/084617 "Dynamic control-to-data transformation to cope with path explosion in symbolic interpretation and/or execution"

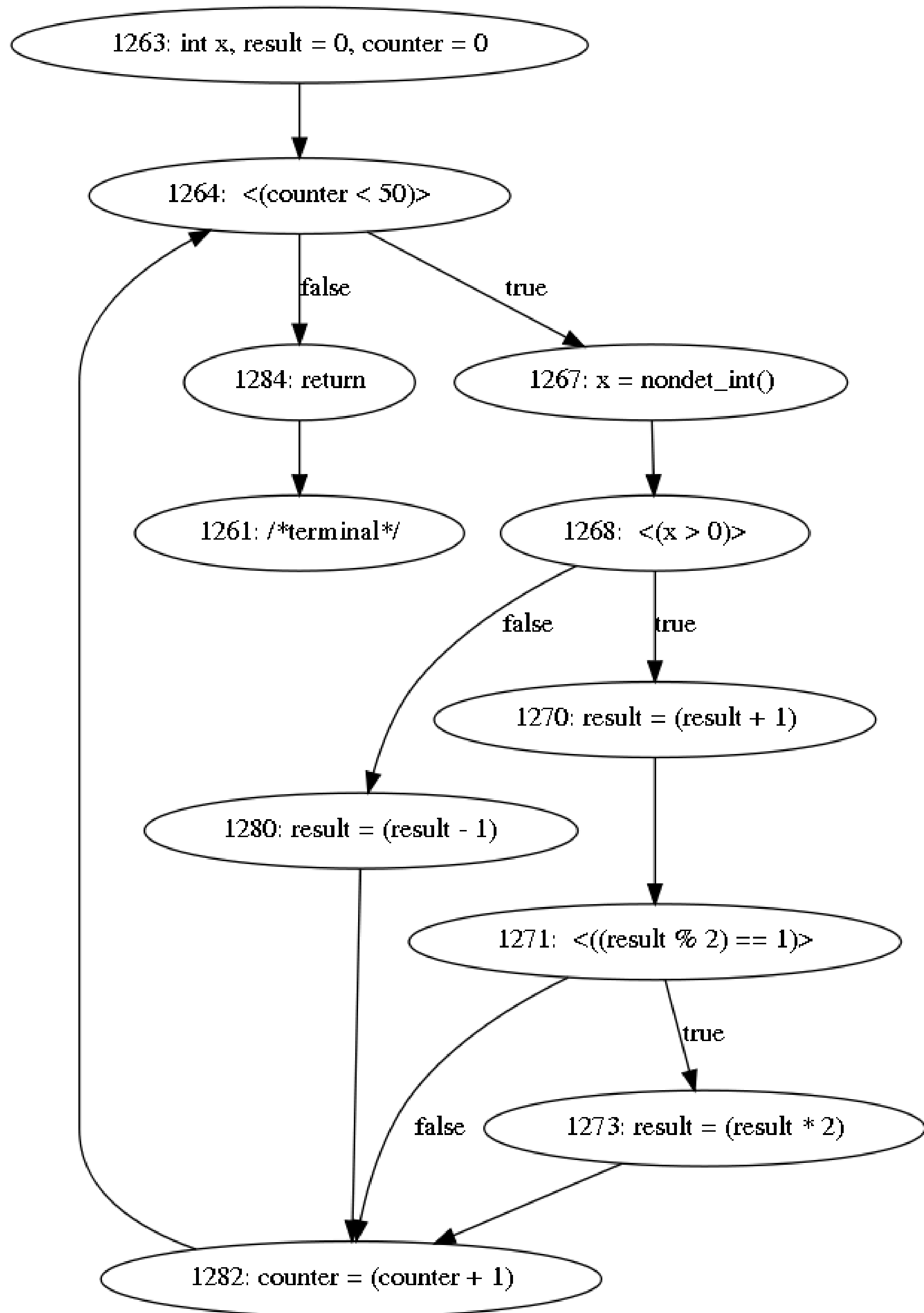

Figure 4.2: CFG of the code demonstrating path explosion problem

In this example, the while loop iterates 50 times, and since it contains two nested if-statements (line 9 and line 11), symbolic execution should traverse nearly 2^{50} paths, which is practically infeasible.

In ExpliSAT, we designed and implemented an algorithm that faces the above problem by avoiding state splitting on if-statements with nondeterministic conditions. For every if-statement where both then-branch and else-branch are feasible, we first find on the CFG a closing node of that if-statement. In our example in Figure 4.2, node 1282 closes the if-statement of node 1268, and it also closes the if-statement of node 1271. After establishing the closing node, the algorithm first traverses then-side of the if-statement, and when reaching the closing node it moves back to the starting node of the if-statement and traverses its else-side. On every traversal, the relevant *context condition* is maintained - the condition of the if-statement for then-side traversal and the negation of this condition for else-side traversal. For nested if-statements the conjunction of their context conditions is maintained. Though the algorithm traverses both then-branch and else-branch of the if-statement, variable assignments under these branches are done only once (either those under then-branch or those under else-branch). To achieve this in our implementation, we convert every assignment into a conditional assignment, where a new value is assigned under the context condition. In other words, we augment the right-hand-side (*rhs*) of every assignment with the context condition, thus forming a conditional rhs expression, where the original rhs appears under the context condition, and the old value of the left-hand side appears under the negation of the context condition.

We illustrate the algorithm run on the CFG in Figure 4.2. The first if-statement (node 1264), resulting from the while loop condition, is always deterministic, since the $counter$ variable is initialized to zero and incremented in every iteration. Thus, at the beginning the algorithm proceeds to the body of the loop. It reaches the if-statement in node 1268, which has a nondeterministic condition $(nondet1 > 0)$, where $nondet1$ is a symbol returned by the first call to $nondet_int()$ function. Then, the closing node (node 1282) of the if-statement is found, and the algorithm proceeds to then-branch, while keeping the context condition $(nondet1 > 0)$. Next, the assignment in node 1270 is processed. Rhs of the assignment is augmented with the condition as follows: $result \leftarrow ((nondet1 > 0)?1 : 0)$. This is because $result$ is initialized to 0, so the old value of $result$ is 0 and its new value is 1. Next, node 1271 is processed. This is an if-statement with condition $result\%2 == 1$. Note, that this condition expression is nondeterministic, because the value of $result$ is nondeterministic expression $((nondet1 > 0)?1 : 0)$. However, under the condition of the outer if-statement $(nondet1 > 0)$ the value of $result$ is 1, and thus the expression $result\%2 == 1$ is simplified to $true$. Thus, the control moves to node 1273, and the value of $result$ is updated as follows: $result \leftarrow ((nondet1 > 0)?(((nondet1 > 0)?1 : 0) * 2) : ((nondet1 > 0)?1 : 0))$. Next, the control reaches node 1282, which is the closing node of the outer if-statement, and the algorithm moves to the else-side. The context condi-

tion is updated to the negation of the if-condition: $!(nondet1 > 0)$. The control reaches node 1280, and $result$ is updated to $result \leftarrow !(nondet1 > 0)?(((nondet1 > 0)?(((nondet1 > 0)?1 : 0) * 2) : ((nondet1 > 0)?1 : 0)) - 1) : ((nondet1 > 0)?(((nondet1 > 0)?1 : 0) * 2) : ((nondet1 > 0)?1 : 0))$. Then the control reaches again node 1282 (the closing node of the if-statement), and if-condition is removed from the context condition, resulting in context condition be $true$. Finally, the algorithm reaches node 1264, and next iteration starts.

We demonstrated how one iteration of the loop is processed within a single path, without state splitting at all. In a similar way all iterations of the loop are processed. Thus, no path explosion happens, and the complexity moves to growing conditional expressions (like the value of $result$ in our example). However, big expressions are efficiently processed by a SAT solver, and the above example completes with our algorithm within a few seconds, whereas it runs into infinity with a classical symbolic interpretation algorithm.

Note, there are cases when branch reduction cannot be performed when keeping semantically equivalent model. For example, given a loop with a nondeterministic condition, the if-statement representing the loop condition cannot be reduced in such a way. The intuition about this problem is that different branches of such an if-statement 'go' to different paths and do not converge in a single node - one side goes back to the beginning of the loop and the other exits the loop. In this case, a loop closing node cannot be identified. Conceptually, if-statements can be classified into two types: if-statements actually changing the control flow (like the while condition descibed above), and those temporarily deviating from the main flow for the sake of different assignments to variables or different function calls. Generally, the former if-statements cannot be reduced by our algorithm, while the latter ones are efficiently reduced.

4.2 Software testing

To perform high-coverage software testing of potentially vulnerable software components, WP4 develops deep fuzzing techniques based on online program analysis. These techniques are applicable to both source and binary programs. In the following, we present the design of the VUzzer tool [69] as the leading software testing technology developed in WP4.

4.2.1 VUzzer

Fuzzing is a testing technique to *catch bugs early*, before they turn into vulnerabilities. However, existing fuzzers have been effective mainly in discovering superficial bugs, close to the surface of software (*low-hanging*

bugs) [34, 29], while struggling with more complex ones. Modern programs have a complex input format and the execution heavily depends on input values conforming to the format. Typically, a fuzzer *blindly* mutates values to generate new inputs. In this (pessimistic) scenario, most of the resulting inputs do not conform to the input format and are rejected in the early stages of the execution. This makes a *traditional random fuzzer* often ineffective in finding bugs that hide deep in the execution.

State-of-the-art fuzzers such as AFL [95] employ *evolutionary algorithms* to operate valid input generation. Such algorithms employ a simple feedback loop to assess how good an input is. In detail, AFL retains *any* input that discovers a new path and mutates that input further to check if doing so leads to new basic blocks. While simple, this strategy cannot effectively select the most promising inputs to mutate from the discovered paths. In addition, mutating an input involves answering two questions: *where* to mutate (which offset in the input) and *what value* to use for the mutation? The problem is that AFL is completely *application-agnostic* and employs a *blind* mutation strategy. It simply relies on generating a huge amount of mutated inputs in the hope of discovering a new basic block. Unfortunately, this approach yields a slow fuzzing strategy, which can only discover deep execution paths by sheer luck. Fortunately, we can increase the efficiency of AFL-like fuzzers manifold by accounting for information that answers the questions above.

In this direction, the use of symbolic and concolic execution has shown promising results [80, 55]. Driller [80] uses concolic execution to enable AFL to explore new paths when it gets stuck on superficial ones. However, fuzzers like AFL are designed to target arbitrarily large programs and, in spite of several advancements, the application of symbolic/concolic techniques to such programs remains a challenge [21]. For example, Driller was benchmarked with 126 DARPA CGC binaries [4] and when AFL got stuck on 41 such binaries, its concolic engine could only generate new meaningful inputs from 13 of such binaries. The results reported in the LAVA paper [34] evidence similar problems with symbolic execution approaches. In another recent study [92], the authors reported that symbolic execution-based input generation (using KLEE) is not very effective at exploring meaningful and deeper paths. In essence, while combining fuzzing with symbolic execution is an interesting research area, this approach also significantly weakens one of fuzzing's original key strengths: *scalability*.

We present VUzzer, an *application-aware* evolutionary fuzzer which is both scalable and fast to discover bugs deep in the execution. In contrast to approaches that optimize the input generation process to produce inputs at maximum rates, our work explores a new point in the design space, where we do more work at the front-end and produce fewer but better inputs. The key intuition is that we can enhance the efficiency of general-purpose fuzzers with a "*smart*" mutation feedback loop based on *control- and data-*

flow application features without having to resort to less scalable symbolic execution. We show that we can extract such features by lightweight static and dynamic analysis of the application during fuzzing runs. Our control-flow features allow VUzzer to *prioritize* deep (and thus interesting) paths and *deprioritize* frequent (and thus uninteresting) paths when mutating inputs. Our data-flow features allow VUzzer to accurately determine *where* and *how* to mutate such inputs. Thanks to its application-aware mutation strategy, VUzzer is much more efficient than existing fuzzers.

Contributions We make the following contributions:

1. We show that modern fuzzers can be "*smarter*" without resorting to symbolic execution (which is hard to scale). Our application-aware mutation strategy improves the input generation process of state-of-the-art fuzzers such as AFL by orders of magnitude.

2. We present several application features to support *meaningful* mutation of inputs.

3. We implement and evaluate VUzzer, a fully functional fuzzer that implements our approach and show that it is highly effective in practice. To foster further research, we will also open source our prototype.

4.3 Memory error defenses

To disarm residual memory error vulnerabilities, WP4 develops both memory error detection and containment techniques, with a focus on efficient and practical solutions. The former offer attack detection guarantees, while the latter are normally more efficient and versatile. In the following, we exemplify this property by presenting the design of two representative memory error defenses developed in WP4: TypeSan [42] and VTPin [71]. Both seek to harden C++ software against memory error vulnerabilities (type confusion and use-after-free respectively), but the former is only applicable to source programs while the latter can directly operate at the binary level.

4.3.1 Memory error detection: TypeSan

Type confusion bugs are emerging as one of the most important attack vectors to compromise C++ applications. C++ is popular in large software projects that require both the modularity of object-oriented programming and the high efficiency offered by low-level access to memory and system intrinsics. Examples of large C++ programs are Google Chrome, large parts of Microsoft Windows and Firefox, and the Oracle Java Virtual Machine. Unfortunately, C++ enforces neither type nor memory safety. This

lack of safety leads to type confusion vulnerabilities that can be abused to attack certain programs. For example, exploitable type confusion bugs have been found in a wide range of software products, such as Adobe Flash (CVE-2015-3077), Microsoft Internet Explorer (CVE-2015-6184), PHP (CVE-2016-3185), and Google Chrome (CVE-2013-0912).

Current defenses against type confusion [10, 52] are impractical for production systems, because they are too slow, suffer from low coverage, and/or only support non-polymorphic classes. The greatest challenge in building an always-on type checker is the need for per-object metadata tracking which quickly becomes a bottleneck if the program allocates, frees, casts, and uses objects at high frequency (e.g., on the stack).

To address the high overhead and the low coverage of existing solutions, we present TypeSan, an explicit type checking mechanism that uses LLVM-based instrumentation to enforce explicit type checks. Compared to previous work, TypeSan provides extended coverage and massively reduced performance overhead. Our back-end uses a highly efficient metadata storage (based on a shadowing scheme with a variable compression ratio) to look up types from pointers. This limits the amount of data written for large allocations (such as arrays of objects) while at the same time supporting efficient and scalable lookups, requiring only 3 memory reads to look up a type. We envision this new type of metadata storage to be also useful for other sanitizers, e.g., to verify memory safety, and we plan to explore further applications in future work.

We primarily envision TypeSan as an always-on solution, making explicit type checks practical for commodity software. Used in attack prevention mode, TypeSan-hardened binaries are shipped to end users and terminate the program on bad casts, thereby preventing zero-day type confusion exploits. Combined with liveness reports for modern software (like the Google Chrome and Mozilla Firefox crash reporters), such a deployment signals the developers about potentially missing type checks. In addition, TypeSan can be used in software testing where TypeSan identifies potential bad casting in the source code. In relaxed mode, TypeSan simply logs all bad casts to scan for underlying vulnerabilities, e.g., when running a test suite. We have implemented a prototype of TypeSan for Linux on top of LLVM 3.9. Our prototype implementation is compatible with large source code bases, such as the SPEC CPU2006 C++ programs and the Firefox browser.

Contributions We make the following contributions:

1. We present a design for high-performance, high-coverage typecast verification on legacy C++ code that is much faster than the state-of-the-art detector with lower memory overhead and orders of magnitude more typecasts.

2. We shows how our design can deliver both nearly complete coverage and performance that is suitable for production usage.

3. We present an automatically generated test suite for typecasting verification to ensure that all different combinations of C++ types are properly handled.

4. We present an open-source implementation of our TypeSan design, available at `https://github.com/vusec/typesan`.

4.3.2 Memory error containment: VTPin

The recent advances in software hardening have undoubtedly made exploitation a challenging craft [68]. Yet, despite the plethora of defenses in place [57], attackers still find ways to compromise essential commodity software, like web browsers [8, 9]. Modern exploits are highly sophisticated and typically leverage a variety of different vulnerabilities to bypass established protections, such as address space layout randomization (ASLR) [63], non-executable memory [15], and sandboxing [2]. To this end, temporal safety errors, and particularly use-after-free vulnerabilities, are becoming a key component of the attackers' arsenal [1, 7]. Interestingly, exploiting use-after-free bugs does not require corrupting memory; instead, an attacker merely needs to utilize dangling pointers, still accessible by a process, for hijacking the control flow.

Temporal safety violations are extremely effective when (ab)used for compromising large C++ programs. Virtual objects contain (at least) one pointer towards a Virtual Table (VTable), which further contains (function) pointers to the implementation of the methods associated with the respective object(s). An attacker can *hijack* the VTable by forcing the vulnerable program to carefully allocate memory with attacker-controlled data; assuming there are still dangling pointers that (now) point to the hijacked VTable, the control flow of the program can be redirected according to the needs of the attacker. Notice that VTable hijacking, through use-after-free, can be combined with other attack vectors for delivering the end-to-end exploit. In fact, in recent Pwn2Own security contests, *all* major web browsers were compromised using exploits that contain a step where VTable hijacking was the *key* attack vector [8, 9].

We present VTPin, a system for protecting software against VTable hijacking in the *least* intrusive way. VTPin works directly with C++ binaries that provide Run-time Type Information (RTTI), does not rely on complex binary analysis or rewriting (often hindering practical deployment [86]), does not interfere with the strategies and policies imposed by the allocator of the protected program, and provides good performance. VTPin *pins* all freed VTable pointers on a safe VTable by instrumenting every `free` call

of a running program. For every `free`, VTPin quickly identifies if it is associated with a virtual object; in case it is, VTPin handles the deallocation accordingly, otherwise the deallocation is forwarded to the program's allocator. VTPin deallocates all space allocated by the object, but preserves its VTable pointers. Additionally, the value of the contained VTable pointer(s) is replaced with the address of a special VTable that VTPin controls. Any dangling pointer, if triggered, can only invoke a method provided by the corresponding safe object. Subsequently, all virtual-method calls due to dangling pointers are not simply prevented, but can also be logged, tracked, and patched.

VTPin follows two strategies for pinning freed virtual objects. If the memory allocator provides reallocation of memory with particular placement, such as the standard GNU allocator, then VTPin pins just the VTable pointer(s) and *frees* the rest of the object. Otherwise, for slab allocators that do not support reallocation with placement, or for objects with multiple VTable pointers, VTPin pins all the VTable pointers and *maintains* all data associated with the freed virtual object. Notice that VTPin handles only the deallocation of objects; all other memory operations, including the allocation of virtual objects, are outsourced to the default allocator of the protected program. VTPin pins only virtual objects, and in most cases only a single pointer (i.e., 8 bytes) survives. Hence, the induced memory overhead is low. In addition, VTPin employs a conservative garbage collector for periodically removing all pinned objects and reclaiming back their occupied memory.

Contributions We make the following contributions:

1. We design and implement VTPin: a system to protect VTable pointers from exploitation through use-after-free vulnerabilities. VTPin does not require access to the source code of the protected program, is not based on binary analysis or rewriting, is highly portable, and does not interfere with the semantics and policies used by standard allocators.
2. We evaluate VTPin with the C++ programs of SPEC, Chromium, and Mozilla Firefox and show that our solution is amenable to practical deployment.

4.4 Exploitation defenses

To prevent attacks based on residual exploitable memory error vulnerabilities, WP4 develops both passive and active exploitation defenses. Passive defenses seek to prevent successful exploitation of a given vulnerability by means of randomization techniques. Active defenses, in turn, operate security checks during the execution to detect exploitation attempts from pro-

gram invariants violations. Our focus in both cases is to improve the state of the art in terms of security and performance, by investigating techniques in different application domains. In the following, we first present the design of the Anti-ROP and sJMP tools as representative of binary-level passive exploitation defenses developed in WP4 and then turn our attention to active defenses.

4.4.1 Passive defenses: Anti-ROP

Return Oriented-Programming (ROP) is an exploit technique where an attacker takes control of a program flow by smashing the call stack, to execute instruction sequences. In the ROP technique, the attacker borrows gadgets (small pieces of code) from the hijacked program to execute the malicious code. ROP is widely adapted by hackers to easily bypass Data Execution Prevention (DEP), which is implemented in hardware and software to prevent code injection and execution. Thus by using ROP, the attackers achieve the same goals achieved by code injection.

Many techniques were developed to prevent ROP attack where Address Space Layout Randomization (ASLR) is implemented almost in all operating system which challenges the attacker to guess the addresses of the gadgets by randomizing the address bases of the sections. There are parts and files that are not randomized which leave weak and soft spots in the program that attackers use to invoke their malicious intentions.

We have developed Anti-ROP, a ROP-prevention system compatible to all platforms and operating system. When building a ROP attack, the attackers assumption is that the gadgets she needs are in absolute addresses or shifted by constant bytes, furthermore, if the attacker detects the shifting for one gadget, then, she detects all gadgets and this is what we smash with our solution, thus, the detecting of the shift of one gadget does not reveal the entire shifting. Our solution can be implemented on Windows and Linux and other operating system, (i) analyzes PE/ELF binary executables, (ii) builds chunks composed of a number of function blocks and (iii) reorganizes them.

Every phase of the process can be invoked online or offline. The randomized file which is created has the same functionality of the original file so that the randomization process requires intervention in the structure of the file, fixing headers, updating branches and tables, etc. As we can see, unlike ASLR technology, finding a gadget in one block, does not reveal the addresses of the other gadgets.

Contributions We have developed a new solution to counter ROP attacks based on fine-grained code randomization. Our solution allows blocking memory corruption Zero-Day attacks, and is able to detect these attack attempts, all with next to zero performance cost.

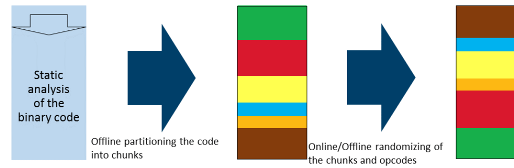

Figure 4.3: Anti-ROP Architecture

4.4.1.1 sJMP

The sJPM (secure jump) technique has two goals: (a) drastically reduce the likelihood of successful ROP attacks, and (b) allow access control policy for library calls to be enforced at the user-code level.

Under our scheme, code space is partitioned into a number of regions – roughly, but not necessarily, isolating each library in a separate region – and controlling access to intra-region jumps. We refer to the latter category as secure jumps or sJMPs.

Each time an sJMP is performed, special policy evaluation code is executed, to determine whether the control flow transfer is warranted (i.e. consistent with the program run-time policy). This allows high-level policy checks to be carried out in a similar fashion to the systrace engine [67], which, however operates at the system call level.

Return-oriented-programming typically relies on locating within the code space of the victim process, sequences that end in a return instruction ("gadgets"). By combining these "gadgets" the attacker can coopt the victim to perform unintended actions. The partitioning of the address space, performed by sJMP, limits the space that attackers may search to locate gadgets. If the attacker attempts to invoke a gadget in another region, the policy engine will be invoked and the attack detected.

sJMP combines two techniques developed locally, namely Instruction Set Randomisation (ISR) [47] and Access Controls For Libraries and Modules (SecMOD) [48]. SecMOD relied on an RPC (remote procedure call) based mechanism to force the user-level code to perform library calls only via the library policy enforcement engine.

The sJMP mechanism developed at TUBS, leverages ISR to assign different encryption keys to each region, so that simply "jumping" to another region will not work since the correct key has to be inserted into the ISR register. Jumps within a region work as before, but inter-region jumps are

monitored. In our first prototype we invoke the kernel, which loads the ISR key for the destination region before executing the jump. This technique, though, will incur some overhead, as a supervisor transition is required for every intra-region jump. Another consequence of this technique is that the RTN instructions will also require a similar check because in this case as well the ISR key will need to be changed.

In order to dispense with the kernel invocation we are developing a technique using new hardware support for “gates” to each region. Under this scheme, when the calling code needs to access a specific remote region, it must go through one of the gates in the destination region to get the correct ISR key loaded. The gate instruction automatically loads the ISR key, without having to invoke the kernel. After the gate, policy evaluation code may be added to implement policy enforcement. The sJMP mechanism will not allow code jumping into the region to bypass the policy check.

Contributions Our contributions are as follows:

1. We have developed a mechanism that leverages ISR to mitigate code reuse / injection attacks.

2. Mandatory policy checks are carried out whenever a library function is called.

This work is in progress and will be later submitted to an academic conference.

4.4.2 Active defenses

To illustrate our approach for active defenses, we now present the design of our pilot CFI technologies tailored to different application domains. PathArmor [88] implements hardware-supported CFI on source or binary programs—using branch tracing facilities provided by commodity hardware. ShrinkWrap [41] implements software-only CFI on source C++ programs. TypeArmor [89] implements software-only CFI for generic binary programs. All these solutions cover the entire spectrum of implementation constraints dictated by the SHARCS models (Chapter 2), with the exception of hardware-only CFI already covered by WP3 (see HCFI in D3.2). Finally, we complement our CFI solutions with a GPGPU-based network intrusion detection system.

4.4.2.1 PathArmor

Control-Flow Integrity (CFI) [13] has developed into one of the most promising techniques to stop code reuse attacks against C and C++ programs. Typically, such attacks circumvent common defenses such as DEP by diverting a program’s control flow to a set of Return-Oriented Programming (ROP)

gadgets [77, 26]. Likewise, they defeat widely deployed ASLR by either targeting gadgets at fixed (non-randomized) addresses [19], or by dynamically disclosing the addresses of randomized gadgets [78]. CFI promises to prevent all such attacks by ensuring that all control transfers conform to the program's original Control Flow Graph (CFG). In theory, CFI is very powerful and, in its purest and ideal form, provably secure against most integrity violations of the control flow [12].

Ten years after the original CFI proposal [13], however, researchers are still working to find practical CFI implementations [97, 96, 87, 51, 27, 36, 62], able to approximate the security of the purest form of CFI with acceptable performance. Common CFI solutions, including state-of-the-art binary-level implementations such as bin-CFI [97] and CCFIR [96], attempt to substantially relax constraints on the set of legal targets for both the backward (e.g., `ret` instructions) and forward (e.g., indirect `call` instructions) control edges. While doing so reduces the performance overhead to a few percent only, it also provides more degrees of freedom for the attackers. Other even more lightweight CFI solutions, such as ROPecker [27] and kBouncer [62], build on heuristics and hardware support to detect anomalous control flows—which resemble ROP gadget chains—and stop many current exploitation attempts at low performance overheads. Unfortunately, a string of recent publications comprehensively shows that it is possible to circumvent all these lightweight CFI solutions with relatively little effort [38, 39, 31, 23, 74].

A key problem with traditional CFI solutions—even recent source-level fine-grained ones [87]—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 [38]).

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 control-flow *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 [13].

We demonstrate that Context-sensitive CFI (CCFI) can indeed be implemented in an efficient, reliable, and practical way for real-world applications. We present PathArmor[3], the first binary-level CCFI solution which

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

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 can be used to mount control-flow diversion attacks [62], 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 ($\overline{C}$CFI) policies.

Contributions Our contribution is threefold:

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

2. We present 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 complement our solution with fine-grained $\overline{C}$CFI policies and simple but comprehensive (backward and forward edge) CCFI policies, making PathArmor the first practical end-to-end CCFI implementation.

3. We show that PathArmor is efficient and can significantly restrict the number of legal control flows compared to traditional CFI solutions.

4.4.2.2 ShrinkWrap

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++ and their *virtual function tables* (or *VTables*) in particular are frequently targeted

by attackers. 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 [13], CFI has seen a long line of followers and variants since [96, 97, 60]. 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.

We here aim to provide the final say on VTable protection, by tightly constraining virtual function pointers (vfp) to the VTables corresponding to the classes intended by the programmer—as defined by the semantics of the C++ language. We examine a recent compiler-based CFI approach, namely VTV [87] and find that its own vfp restrictions are not entirely accurate.[4] We proceed by extending VTV to apply even tighter restrictions to the available VTable targets and argue that our approach is optimal. More precisely, we aim at offering the best protection possible to vfps in a context insensitive fashion. We build our solution into VTV and evaluate it by means of a new framework for testing it. Using the framework, we show that it is the strictest access policy possible for VTables without breaking legitimate code. Last but not least, our solution is *faster* than the original VTV implementation.

Our investigation exposes *three* weaknesses of existing schemes. First, we find 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 find that even state-of-the-art solutions, like VTV, 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 identify 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.

We use our observations to design ShrinkWrap, a new VTable-protection scheme that uses information available during compilation to extract the most restrictive set of VTables that should accessible at a virtual call-site within the code. We implement this enhanced, fine-grained design on top

[4] VTV is now a standard compiler option also used in production systems.

of VTV and evaluate it by creating a testing framework that exhaustively explores all possible combinations of class inheritance and method invocation to demonstrate that our technique provides the best possible defense for VTables. We also demonstrate that our modifications do not break complex, real-life applications and that our solution is faster than the original VTV.

Contributions We summarize our contributions below:

1. We identify limitations in the design and implementation of current VTable protection schemes, including the primary industrial implementation, VTV.

2. We identify 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.

3. We develop a practical testing methodology to evaluate VTable protection schemes and to highlight potential limitations.

4. We implement ShrinkWrap, a prototype of the proposed protection scheme and show it is more secure and efficient than state-of-the-art solutions.

4.4.2.3 TypeArmor

Control-Flow Integrity (CFI) [13] is one of the most promising ways to stop advanced code-reuse attacks. Unfortunately, enforcing it without access to source code is hard in practice. The reason is that it requires an accurate Control-Flow Graph (CFG) and extracting such CFG from binary code is an undecidable problem. As a result, most existing binary-level CFI implementations base their invariants on an *approximation* of the CFG which leaves enough wiggle room for an attacker to launch successful exploits [23, 26, 31, 38, 39, 74].

While it may be possible to stop some advanced attacks using a perfect shadow stack implementation [22], there is one class of attacks for which there is no existing defense at the binary level whatsoever. This class of *function-reuse* attacks, exemplified by Counterfeit Object-Oriented Programming (COOP) [73], chains together calls to *existing* functions through *legitimate* callsites. This strategy preserves the integrity of the shadow stack, while abusing the overapproximation of the extracted CFG to call the *wrong* functions from these callsites. This attack is powerful since it allows for exploits that integrate smoothly with legitimate code execution. Unless there is deep knowledge of the C++ class hierarchy semantics, which we can

only extract if we have the source code [30], it is hard to tell a COOP exploit apart from a legitimate code sequence [73]. Lacking a handle on the functions that a callsite may target leaves all the existing binary-level CFI solutions unable to stop COOP attacks.

We revisit binary-level protection in the face of COOP attacks and follow-up improvements [30]. We explore to what extent we can narrow down the set of possible targets for indirect callsites and stop exploitation at the binary level. Our conclusion is *not* that all possible attacks can be stopped: even the tightest CFI solutions *with* access to source code are unable to guarantee perfect protection against all possible attacks [22]. Nevertheless, we demonstrate that TypeArmor, our binary-level protection prototype, can stop all COOP attacks published to date and significantly raise the bar for an adversary. Moreover, TypeArmor provides strong mitigation for many types of code-reuse attacks (CRAs) for programs binaries, without requiring access to source code. As researchers have shown that it is easy to bypass existing binary-level CFI defenses [23, 26, 31, 38, 39, 74], this is a significant improvement.

TypeArmor incorporates a forward-edge CFI strategy that relies on conservatively reconstructing both callee prototypes and callsite signatures and then uses this information to enforce that each callsite *strictly* targets matching functions only. For example, TypeArmor disallows an indirect call that prepares fewer arguments than the target callee consumes. Additionally, TypeArmor incorporates a novel protection policy, namely CFC (*Control-Flow Containment*), which further reduces the possible target set of callees for each callsite. CFC is based on the observation that, if binary programs adhere to standard calling conventions for indirect calls, undefined arguments at the callsite are not used by *any* callee by design. TypeArmor trashes these so-called *spurious arguments* and thus breaks all published COOP and improved COOP-like exploits. These exploits all chain virtual method calls that disrespect calling conventions to achieve convenient data flows between gadgets [30]. CFC eliminates these data flows via unused argument registers and thus stops such exploitation attempts.

Current binary-level solutions enforce “loose” forward-edge CFI policies, often allowing control transfers from any valid callsite to any valid referenced entry point [97, 96]. In the best case, existing policies only reduce the target set by removing all entry points of other modules unless they were explicitly exported or observed at runtime [64]. In contrast, TypeArmor matches up indirect callsites with a more precise target set in a many-to-many relationship. It relies on use-def analysis at all possible callees to approximate the function prototypes, and liveness analysis at indirect callsites to approximate callsite signatures. This effectively leads to a more precise CFG of the binary program in question, which could also be used by existing mitigation systems to amplify their (context-insensitive) invariants (e.g., PathArmor [88]).

Contributions We make the following contributions:

1. We introduce techniques to recover callsite signatures and callee prototypes for security enforcement purposes. Our techniques yield binary-level control-flow invariants which approximate the type-based invariants used in source-level solutions [87] and are thus much more precise than those used in prior binary-level CFI solutions [97, 96, 64].

2. We demonstrate that fully-precise, static forward-edge CFI is inherently hard to achieve in a conservative fashion, due to the unavoidable precision loss when deriving traditional CFI-style target-oriented invariants at the binary level. To compensate for the precision loss, we complement our CFI strategy with a new technique termed *Control-Flow Containment* (CFC). CFC relies only on our callsite analysis to effectively *contain* code-reuse attacks. This approach improves the quality of control-flow invariants of traditional target-based approaches, overall resulting in a strict binary-level CFI strategy.

3. We implement TypeArmor, a new strict CFI solution for program binaries. We show that TypeArmor is efficient and can enforce much stronger forward-edge invariants than all the existing binary-level CFI solutions.

4. We show that our strict binary-level CFI strategy can mitigate advanced attacks in complete absence of source information or C++ semantics. For example, TypeArmor can stop all published COOP [73] exploits and their improvements [30].

4.4.3 GPGPU network intrusion detection systems

Network Intrusion Systems act as a front line defense in order to filter incoming traffic for malicious activity. In cloud infrastructures the amount of traffic that needs to filtered is massive. A proposed solution to the extended network packet processing requirements of such infrastructures, is to offload the packet processing to commodity hardware such as General Purpose Graphics Processing Units. GPUs are specialized for intensive highly parallel computations like signature matching. Additionally, GPGPUs are relatively cheap and widely used in large server infrastructures as co-processors.

In order to use GPGPUs for network intrusion detection efficiently we have developed the following software pipeline. A packet acquisition module collects all the incoming packets inside a ring buffer. Every time the buffer is full the batch of packets is transfered to the GPGPU. The GPGPU scans the batch of packets for matching rules and returns the results back to host. The host is responsible to report any malicious event detected. A batch

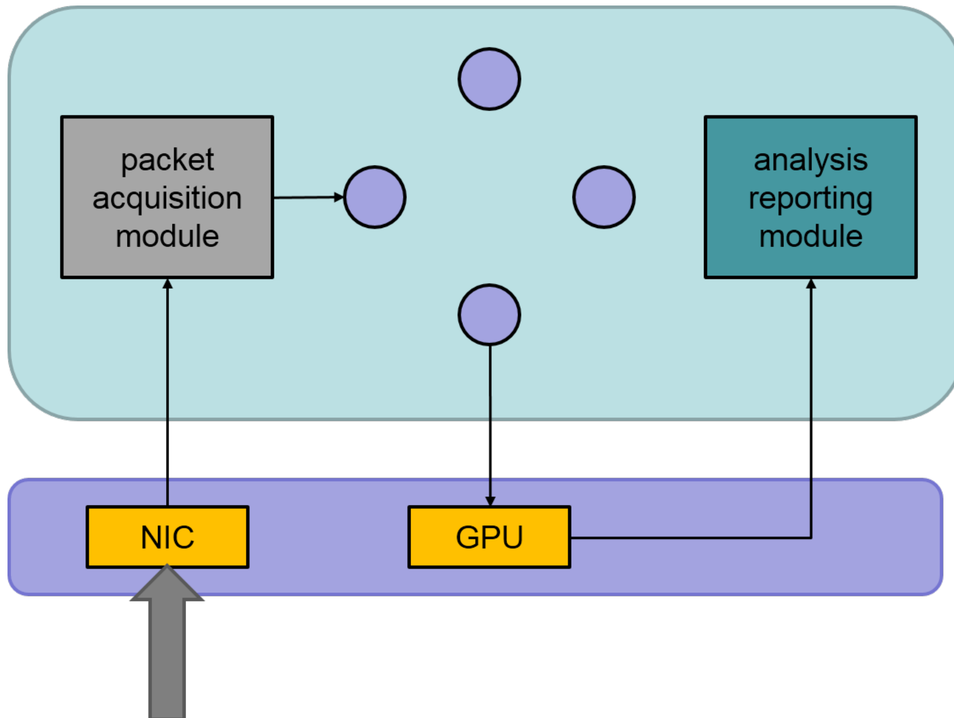

Figure 4.4: The GPGPU NIDS architecture

of packets instead of streaming is important in order to reduce memory traffic between host and device memory and maximally utilize the GPGPU.

This work is in progress and will be later submitted to an academic conference.

4.5 Integrity monitoring

To monitor software execution for malicious changes operated by a successful exploit, WP4 develops efficient snapshot-based integrity monitoring techniques based on commodity hardware support. These techniques are applicable to both source and binary programs. We specifically focus on kernel integrity monitoring techniques, since real-world rootkits that operate persistent malicious changes to systems typically target the operating system kernel. In the following, we present the design of the GRIM system [50] as the leading integrity monitoring technology developed in WP4.

4.5.1 GRIM

Despite the recent advances in software security, vulnerabilities can still be exploited if the adversary is really determined. No matter the protection

enabled, there is always a path for successful exploitation, although admittedly, today, following this path is much harder than it was in the past. Since securing software is still under ongoing research, the community has investigated alternative methods for *protecting* software. One of the most promising is monitoring the Operating System (OS) for possible exploitation. Once an abnormality is detected then the monitor should be able to terminate the system's operation and alert the administrator.

This form of protection is commonly offered by tools known as *Kernel Integrity Monitors* (KIMs). The core operation of these tools is to inspect, as frequently as possible, both the kernel code and data for determining if something has been illegally modified. In principle, compromising an operating system is usually carried out using a kernel rootkit (i.e., a piece of malicious code that is installed in the OS), which usually subverts the legitimate operation of the system by injecting malicious functionality. For example, the simplest way for achieving this is by inserting a *new* (malicious) system call, which, obviously alters a fundamental structure in the kernel's code: the system-call table. In order to identify such a simple rootkit, it is enough to only monitor the memory region where the system-call table is mapped for possible changes.

Implementing KIMs may sound trivial, however the level of sophistication of modern kernel rootkits, gives space for many different choices. A straightforward approach is to implement the monitor solely in software, in the form of a hypervisor which runs and frequently introspects the OS for possible (malicious) changes [16, 45, 91, 76]. This choice is really convenient, since there is no need for installing custom hardware, nevertheless it is implied that the monitor's code is non vulnerable itself. Unfortunately, it has been demonstrated that a hypervisor can be compromised by code running at the guest OS [70]. In addition, formally verifying the monitor's code may need significant effort [49]. A viable alternative is to offer monitors that are implemented in hardware. Copilot [65] is a representative architecture, implemented in a PCI card and it is basically a snapshot-based monitor. Essentially, a snapshot-based system monitors a particular memory region to identify possible malicious changes. As an example, consider the simple case of monitoring the region where the system-call table has been mapped, in order to detect if a new (malicious) system call has be injected. Copilot has a transparent operation, allowing the OS to be unaware of its existence, and thus it stands as a very attractive option, especially in terms of deployment. Still, modern rootkits have evolved and developed techniques that can evade detection, by exploiting the window of opportunity between two snapshots. As a matter of fact, a rootkit can simply perform a (malicious) change right after a snapshot is taken and subsequently remove it before the next snapshot is available.

To overcome this inherent limitation of snaphost-based detection systems, recent proposals have been focused on *snooping* based detection [58,

53]. A snoop-based system monitors all the operations that are sent over the memory bus. In the context of the aforementioned example we used, the snoop-based detector would have achieved equivalent detection with the snapshot-based system by capturing the *write* operations that aim at modifying the region where the system-call table is mapped. It is evident, that the snoop-based approach performs a lighter check, since instead of monitoring a particular region, it monitors the bus for a particular operation. Nevertheless, snooping is possible *only* in custom processors, since the memory controller is integrated to the CPU, which poses critical deployment issues. The benefits of snoop-based systems were introduced by Vigilare [58] and have been demonstrated in KI-Mon [53] where, in short, the authors provide experimental evidence that a snapshot-based approach can only reach 70% of detection rate, while their snoop-based system, KI-Mon, can reach 100% of detection rate.

Although we acknowledge the benefit of snoop-based systems, such as KI-Mon, we stress that snapshot-based systems can essentially *do better*. We implement GRIM, a novel snapshot-based KIM based on a GPU architecture. Using GRIM we can easily reach 100% of detection rate using a snapshot-based *only* architecture. GRIM promotes the design of a novel architecture, which *does not only* demonstrate high detection rates in snapshot-based integrity monitors, but, also, provides a generic extensible platform for developing KIMs that can be instantly deployed. GRIM works transparently and requires no modifications such as re-compilation of the kernel and installing custom hardware on the system it protects. In addition, GRIM does not aim at simply promoting the usage of GPUs in a new domain. To the contrary, GRIM delivers a design that demonstrates many critical properties for KIMs. Beyond the dramatic speed gains in detection rates, the system is easily programmable and extensible, while it is not based on custom hardware but on commodity devices.

Contributions We make the following contributions:

1. We design and implement GRIM, a novel GPU-based kernel integrity monitor.

2. GRIM demonstrates that snapshot-based monitors can do substantially better than it has been so far documented in current literature [53]. We are able to reach easily 100% detection rate, surpassing substantially the reported detection rate (70%) in the state of the art.

3. GRIM is fully programmable and it provides a generic extensible platform for developing KIMs that can be instantly deployed using just commodity devices. It works transparently and requires no modifications such as re-compilation of the kernel and installing custom hardware on the system it protects.

4.6 Recovery

To recover the execution in a safe state after a successful or unsuccessful exploitation attempt—or more generically a software failure, possibly triggered by a bug even during attack-free execution—WP4 develops automated error recovery techniques based on efficient memory checkpointing. These techniques are applicable to both source and binary programs. In the following, we present the design of the OSIRIS system [18] as a representative recovery technology developed in WP4. OSIRIS focuses on supporting short error recovery windows in the operating system.

4.6.1 OSIRIS

A decade ago, Chou et al. [84, 28] identified buggy kernel extensions as an important cause of operating system failures for commodity operating systems such as Windows and Linux. Assuming that drivers are mostly stateless and faults are mostly transient, we can handle such faults by isolating the driver code and restarting it in case of a failure [44].

However, a more recent study on the Linux kernel [61] shows that faults in stateful core OS subsystems have started to outrank the buggy drivers in importance, even though the latter are still large in number. Furthermore, these bugs are typically *not* transient [25, 40]. In other words, relying on simple re-execution is no longer a viable solution to recover from such faults. For this reason, we seek a method to mitigate their effects.

There are many approaches that improve error recovery in operating systems. Typically, they protect either applications [33] or specific OS subsystems from the effects of software faults in the operating system [82, 83, 46, 98], but there also have been efforts to provide whole-OS error recovery [59, 32, 54, 44].

Extending error recovery techniques to an entire OS is a complex problem. Most of the solutions compartmentalize the OS, primarily to prevent the effects of faults in one component from spreading onto other components. Moreover, compartmentalization allows components to be recovered individually. However, stateful runtime interactions among components make *per-component* recovery nontrivial. In a scheme that enforces strict fault isolation between OS components, recovering from failures resembles a distributed systems recovery problem. In this context, the solutions are generally of three kinds:

1. *Global replication* – which is least suitable for a general-purpose operating system that aims for judicious and efficient usage of system resources;
2. *Dependency tracking* – which does not scale for the high-frequency inter-component interactions found in an operating system setting;

3. *Global checkpointing* – which not only hinders normal execution performance, but also degrades it exponentially when we scale to a larger number of active system components.

Global checkpointing in the context of operating systems [79] offers strong global consistency guarantees, but suffers from the need to synchronize all components, introducing bottlenecks which greatly affect overall system performance. Since OS components interact all the time, checkpoints should be taken at high frequencies—typically orders of magnitude beyond what is possible with today's global checkpointing solutions. Local, per-component checkpointing allows for more concurrency in the system but dependent components still have to coordinate to ensure that their locally checkpointed state remains consistent with that of their counterparts. In case of uncoordinated checkpoints, there is the risk of a *domino effect* [81], where error recovery leads to unbounded rollback of inter-dependent components. In general, expensive runtime dependency tracking is the price that local checkpointing schemes pay towards guaranteeing globally consistent recovery.

We introduce a new design called OSIRIS (OperatingvSystem with Integrated Recovery preventing Inconsistent State)[5], which seeks to strike a new balance between performance and globally consistent recoverability of an operating system.

Building on efficient in-memory checkpointing, OSIRIS recovers only in cases where we can conservatively infer that performing local recovery will not lead to global state inconsistencies. This eliminates the need for dependency tracking and synchronization, also greatly simplifying the recovery mechanism. Such solution offers better performance than any other existing scheme, at the cost of not being able to recover in every case.

Our solution is the first to achieve globally consistent recovery of stateful core OS components with very low performance overhead. OSIRIS brings OS recoverability (limited, but still powerful) within the reach of production systems. Moreover, unlike many existing OS recovery approaches, we do not limit ourselves to drivers and explicitly target the core system services. Given their heavily stateful nature, it is much harder to recover from faults in such components than from faults in drivers. For instance, a system call like `exec` involves the file system, memory manager, cache manager, process manager, etc. Error recovery in any of these components must keep its state consistent with other components' state and resume execution in a globally consistent way.

Our approach uses knowledge about the nature of inter-component interactions in the system to perform recovery only within dependency-safe *recovery windows*—intervals during which state changes within a component have not affected other components. We use a lightweight in-memory

[5] OSIRIS is open source, available via `https://github.com/vusec/osiris`

checkpointing system [90] to allow efficient high-frequency creation of per-component checkpoints. We further optimize this approach by disabling our runtime (memory logging) instrumentation whenever recovery to a consistent state is known to be impossible. Our design results in lower runtime overhead and provides a fine-grained trade-off between recoverable surface and performance. Finally, instead of replaying the execution after recovering from a failure, we send an error to the component that sent the request that triggered the failure. This allows us to deal with persistent faults in addition to transient faults.

Contributions We make the following contributions:

1. We describe an operating system recovery method that determines whether rolling back only component-local state can restore the OS to a globally consistent state.
2. We show how OSIRIS occupies a meaningful new point in the design space of recoverable operating systems by introducing a new trade-off that solves the performance, maintenance, and complexity drawbacks of existing solutions at the cost of reduced recovery surface.
3. We explain how our approach deals with persistent software faults in core system services whereas most prior efforts are limited to transient faults.
4. We minimize checkpointing instrumentation overhead by disabling memory logging when we cannot recover, providing a fine-grained trade-off between performance and recovery surface.
5. We show that our prototype implementation is efficient performs meaningful recovery from real-world faults.

4.7 Web and mobile security and privacy

To enhance the security of the user-end applications, WP4 develops techniques targeting Web Browsers, Android and iOS mobile devices and also new novel approaches of secure communication procedures. These techniques are implemented in the source code of the user-end application, or perform analysis on the applications runtime.

4.7.1 CCSP

Trust in TLS-based communication on the Internet is provided by Certificate Authorities in the form of signed certificates. Browsers check the validity of the certificates they receive from websites via the Online Certificate

Status Protocol(OCSP), or via Certificate Revocation Lists(CRLs). Unfortunately, sophisticated cyber-attackers who manage to act as men-in-the-middle, may trick browsers to trust revoked certificates, believing that they are still valid. Consequently, the browser will communicate (over TLS) with servers controlled by cyber-attackers. Although frequently updated, nonced, and timestamped certificates may reduce the frequency and impact of such cyber-attacks, they impose a very large overhead to CAs and OCSP servers, which now need to timestamp and sign on a regular basis all the responses, for every certificate they have issued. To mitigate this overhead and provide a solution to the described cyber-attacks, we have developed CCSP: a new approach to provide fresh information regarding certificates' status, which capitalizes on a newly introduced notion called signed collections. Our preliminary results suggest that CCSP

Contributions We make the following contributions:

1. Reduce space requirements by more than an order of magnitude.
2. Lower the number of signatures required by several orders of magnitude compared to OCSP-based methods.
3. Add only a few milliseconds of overhead in the overall user latency.

This work is accepted for publication at the IEEE International Conference on Computer Communications 2017.

4.7.2 Constant blinding in JIT engines

In this work we evaluate the security levels provided by modern JavaScript just-in-time (JIT) engines that are integrated in web browsers, for example SpiderMonkey (Mozilla Firefox) and Chakra (Microsoft Edge). Specifically, it has been proven that malware authors can utilize the JIT engine to generate their own sequences of instructions, which combined with other browser bugs can allow arbitrary code execution when the user visits the malicious page. This is done by inserting specially crafted constant values in the JS code of the web page. We are evaluating the effectiveness of the mitigations utilized by the latest JIT engines, as well as attempt to harden the JIT engines by extending or introducing new countermeasures. In order to reduce the risk of constant values that can be used to execute arbitrary code, we develop constant blinding (or constant encoding), which randomizes the values emitted in the code buffer during every run, thus making it impossible for the attacker to exploit.

This work is almost finished and will be submitted to an academic conference.

4.7.3 Proxy injection detection

Internet users often place their trust on systems that are not under their control. From a security and privacy perspective, a particularly critical class of such systems is HTTP proxies that act as “stepping stones” between clients and servers. By relaying their traffic through a proxy, users can access otherwise blocked content and services due to geographical restrictions, content filtering policies, or censorship, and to some extent preserve their anonymity, by hiding the originating IP address from the final destination (although still exposing it to the proxy).

Once the user traffic reaches the proxy towards its actual destination, unless end-to-end encryption is used, a rogue or compromised proxy can tamper with the transmitted content or snoop for sensitive user data [72]. Even when end-to-end encryption is used, however, the problem is not alleviated, as man-in-the-middle attacks are still possible using fake or even valid—obtained through compromised CAs or generated by powerful adversaries—certificates, or SSL-stripping attacks [56]. The potential harm due to network traffic interception attacks can be severe, ranging from mere annoyance and inconvenience to system compromise and theft of private information.

The proliferation and widespread use of web proxies necessitate the need for understanding and measuring the extent of content modification by rogue web proxies. The ease of setting up a free online proxy (e.g., on a cloud-hosted virtual machine) and registering it on the numerous “proxy list” websites, raises the question of whether miscreants employ these tactics to attract and gain access to user traffic, compromise user devices, or steal private information. Sporadic evidence so far has shown that this is indeed happening in various types of network relays, including VPN servers and anonymity network relays [94, 66, 43, 93, 24, 11], but the extent of the problem in the front of web proxies remains unknown.

As a step towards understanding and measuring the extent of content modification by rogue HTTP proxies, we have conducted a large-scale study of open HTTP proxies, focusing on the detection of content alteration or injection events. We periodically collected publicly available proxies by crawling a set of “proxy list” websites, and probed them using a novel technique based on *honeysites* under our control. Instead of visiting real web sites with and without a proxy, and comparing the difference in the retrieved content, the use of honeysites allows us to avoid any false positives issues due to highly dynamic content (e.g., news updates, rotating ads, and time-related information), or content localization (e.g., due to the proxy’s different location than the client). We employed a set of honeysites of increasing complexity and content diversity, and implemented a content modification technique that operates at the level of a page’s DOM tree to detect even slight object modifications.

After successfully probing 9,412 proxies, our results indicate that 739 of them (7.85%) performed some form of malicious or unwanted content modification. The observed alterations included the injection of extra (or the modification of existing) ads, the inclusion of tracking libraries and fingerprinting code, the collection of data from social networking services on which the user is already authenticated, and the injection of malicious code. Besides the injection of ads and user tracking and fingerprinting code, we also discovered more severe and sophisticated instances of malicious behavior, such as XSS attacks, SSL stripping, and redirection to servers that have been reported to host malware.

Contributions We make the following contributions:

- We present an approach for the detection of unwanted or malicious content modification by rogue HTTP proxies, based on the observation of discrepancies in the retrieval of content through the probed proxies. Our technique uses a set of decoy web sites, dubbed *honeysites*, which are created with an incremental degree of complexity and external content dependencies, to allow differentiating between different types of content injection or alteration.
- We have performed a large-scale measurement study of open HTTP proxies retrieved from public "proxy list" websites, and used our technique to assess the extent of malicious content modification by rogue proxies. Our findings suggest that 7.85% of the tested proxies engaged in some form of malicious or unwanted content modification.

This work is currently under submission to the ACM Asia Conference on Computer and Communications Security 2017.

4.7.4 Sandboxing mobile application on the cloud

Many applications make use of the capabilities the cloud has to offer. In this work we developed a system were applications operate at the cloud while users interact with them through chrome remote desktop. We selected skype as the main application being tested since it is one of the most popular messaging and video calling applications and also one of the most challenging to operate exclusively on the cloud. Furthermore for a High Definition video calling experience between two users, many conditions must be taken into account . The most common conditions that may influence an HD video call is the available bandwidth, the cpu power and the available ram. Skype, amongst other factors, can determine the bandwidth available as well as the PC specs and drop or increase the quality of the video call accordingly. Even if there is a strong internet connection, skype may drop the video call

resolution, if the cpu usage reaches a certain percentage. Therefore it is recommended to use certain webcams with the ability to encode HD video using there own hardware, in order to avoid stressing the cpu. Our system is based on executing skype in a remote server operating in a high speed network and with sufficient computing power. Thus the main requirements for an HD video call are off loaded from the end user to the server. If both end-users operate Skype remotely, then the actual video call happens between machines with enough computing power and strong internet connection. Since the skype communication does not suffer from congestion, packet loss or limited computing resources, the end-user has a better video calling experience. Finally, since applications are executed on the cloud, any malicious intent(e.q buffer overflows, ROP attacks, IP denial of service, etc.) does not affect the end user. Thus, such a system can be incorporated by cloud companies in order to sandbox their applications and most important provide privacy as a service.

This work is in progress and will be later submitted to an academic conference.

4.7.5 Anti-track droid

In this work we developed an anti-tracking mechanism to enable the users access an online service through a mobile app with- out risking their privacy. Our evaluation shows that our approach is able to reduce the leaking identifiers of apps by 27.41% on average when it imposes a practically negligible latency of less than 1ms per request while preserves the privacy of the user.

In our work, we consider as privacy-related leaks, information, such as (i) installed applications, (ii) known SSIDs, (iii) connected wifi, (iv) kernel characteristics, (v) model version, (vi) carrier, etc. This device-specific information is able to persistently track mobile users without using any deletable cookies or resettable Advertising IDs [35]. By deploying-device fingerprinting a third party can also: (i) decloak user's anonymous sessions: by linking for example Tor sessions of the same device with eponymous ones [14] and (ii) link web with app sessions, one of the biggest challenges of mobile ad networks. Surprisingly, our results show that in case of device-specific privacy leaks, there is a clear winner: mobile browsers leak significantly less information compared to mobile apps. In all the cases we studied, mobile apps leaked tons of information that mobile browsers did not (or could not) leak. Web sites may provide poor functionality to mobile devices and thus, the use of apps may seem the only reasonable choice from a user experience point of view. To improve the privacy of users who must use mobile apps, we developed Anti-Track Droid, an anti-tracking mechanism for mobile apps, tantamount to the current state-of-the-art ad- blockers of mobile browsers. Our approach constitutes an integrated monitoring and filtering module, which contrary to alternative approaches works solely in the users

device without requiring any additional infrastructure (i.e. proxy or VPN). Our evaluation shows that Anti-Track Droid is able to reduce the leaking identifiers of apps by 27.41 , and it imposes an insignificant latency of less than 1ms per request.

Contributions We make the following contributions:

1. We design Anti-Track Droid: a novel anti-tracking mechanism for mobile devices. Similar to state-of-the-art browser ad-blockers, our approach blocks any possible request may deliver to third parties data that can be used either for user profiling or device fingerprinting.

2. We implement our system as an integrated filtering module for Android. Anti-Track Droid uses a mobile-based blacklist, which we publicly release, and it does not require changes in the respective OS or any kind of external infrastructure (i.e. proxy). We experimentally evaluate our prototype and show that in addition to improving privacy, it reduces the bytes transfered by an average of 8%, and it reduces latency by as much as 90% for blacklisted trackers.

This work is currently under submission to the World Wide Web 2017 conference.

4.7.6 REAPER

The Android OS, with each major revision, moves forward towards a more fine grain permission control system. With the newly Android 6 permission system, users are now able to accept or reject a permission request at run time, as well as, revoke dangerous permissions any time through the system settings. So far, the Android ecosystem and its applications are dominated by third party libraries that leak personal information without the users knowledge and concern. Even though such a permission control system gives more power to the user, it still fails to distinguish which part of the application is responsible for the permission request. Motivated by the above rationale, we developed a fine grain permission access control system, were users are fully aware about the permissions needed by the integrated third party libraries. Such a system can be adopted by Google Play, the official Android market and inform users at installation time about the permissions needed by the integrated third parties.

Contributions We make the following contributions:

1. Based on the necessity to identify which parts of an application require certain permissions, we develope Reaper. Reaper is a real-time permission analysis system, based on the Xposed framework, that is capable

to understand which part of the application, (core-functionality/3rd party libs), is requesting access to resources being held by a specific permission. Unlike previous static approaches, our system performs dynamic analysis on applications, without any code modification to either apps or the OS and reveals the permissions requested by the third party.

2. We designed a UI automator prototype to traverse the graph of each application based on a certain threshold. Our sophisticated prototype, harvests only the interactive elements that are drawn at any time on the display without the need of modifying or repackaging the Operating System and the applications.

3. We implemented Reaper as a plug and play framework for any version of the Android OS that supports the Xposed framework and evaluated apps from the official Android Market, Google Play.

This work is in progress and will be later submitted to an academic conference.

5 Conclusion

This deliverable described the core design specification of the SHARCS software technologies. To develop our design, we followed a top-down approach, starting from the requirements and threat model defined in D2.1, and detailing attack primitives, vulnerabilities, and ultimately security defenses to address the threats considered by SHARCS.

Rather than focusing on narrow security defenses, WP4 advances the state of the art with a broad portfolio of security defenses at different design layers, ranging from software verification to exploitation defenses and beyond. This holistic approach is crucial to devise *comprehensive*, *versatile*, *generic*, and *multilevel* solutions, which satisfy our goals of providing end-to-end and in-depth security, and are sufficiently flexible to be applicable to the different SHARCS applications.

Since the techniques developed by WP4 have broad applicability, they can be mapped and used in the systems considered in WP5 in the most flexible way. This is also important to satisfy more generic application requirements (e.g., power consumption) defined in D2.1. For more details on the integration of (a subset of) WP4 techniques for each of the SHARCS applications and systems, we refer the reader to the WP5 deliverables.

Bibliography

[1] Advanced Exploitation of Mozilla Firefox Use-after-free Vulnerability (MFSA 2012-22). http://www.vupen.com/blog/20120625.Advanced_Exploitation_of_Mozilla_Firefox_UaF_CVE-2012-0469.php.

[2] Chromium OS Sandbox. https://www.chromium.org/developers/design-documents/sandbox.

[3] Cve data feed. https://nvd.nist.gov/download.cfm#CVE_FEED.

[4] Darpa cyber grand challenge binaries. https://github.com/CyberGrandChallenge.

[5] The exploit database. https://github.com/offensive-security/exploit-database.

[6] Memory errors tool. https://github.com/vvdveen/memory-errors.

[7] Pwn2Own - Google Chrome Blink Use-After-Free Remote Code Execution Vulnerability. http://www.zerodayinitiative.com/advisories/ZDI-14-086/.

[8] Pwn2Own 2015: The year every web browser went down. http://www.zdnet.com/article/pwn2own-2015-the-year-every-browser-went-down/.

[9] Pwn2Own 2016: Hackers Earn $460,000 for 21 New Flaws. http://www.securityweek.com/pwn2own-2016-hackers-earn-460000-21-new-flaws.

[10] Undefined behavior sanitizer. https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer.

[11] vpwns: Virtual pwned networks. In *Presented as part of the 2nd USENIX Workshop on Free and Open Communications on the Internet*, 2012.

[12] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. A theory of secure control-flow. In *ICFEM*, 2005.

[13] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations, and applications. *TISSEC*, 2009.

[14] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Preneel. Fpdetective: Dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1129–1140, New York, NY, USA, 2013. ACM.

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

[16] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *CCS*, 2014.

[17] S. Barner, C. Eisner, Z. Glazberg, D. Kroening, and I. Rabinovitz. ExpliSAT: Guiding SAT-based software verification with explicit states. In *HVC*. 2007.

[18] K. Bhat, D. Vogt, E. van der Kouwe, B. Gras, L. Sambuc, A. S. Tanenbaum, H. Bos, and C. Giuffrida. OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems. In *DSN*, 2016.

[19] E. Bosman and H. Bos. Framing signals—A return to portable shellcode. In *S&P*, 2014.

[20] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[21] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 2013.

[22] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[23] N. Carlini and D. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *USENIX Security*, 2014.

[24] S. Chakravarty, G. Portokalidis, M. Polychronakis, and A. D. Keromytis. Detecting traffic snooping in Tor using decoys. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 222–241, September 2011.

[25] S. Chandra and P. M. Chen. Whither generic recovery from application faults? a fault study using open-source software. In *DSN*, 2000.

[26] S. Checkoway, L. Davi, A. Dmitrienko, A. R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *CCS*, 2010.

[27] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *NDSS*, 2014.

[28] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP*, 2001.

[29] S. Clark, S. Frei, M. Blaze, and J. Smith. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *ACSAC*, 2010.

[30] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *CCS*, 2015.

[31] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security*, 2014.

[32] F. M. David, E. Chan, J. C. Carlyle, and R. H. Campbell. Curios: Improving reliability through operating system structure. In *OSDI*, 2008.

[33] A. Depoutovitch and M. Stumm. Otherworld: giving applications a chance to survive os kernel crashes. In *EuroSys*, 2010.

[34] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *S&P*, 2016.

[35] P. Eckersley. How unique is your web browser? In *Proceedings of the 10th International Conference on Privacy Enhancing Technologies*, PETS'10, pages 1–18, Berlin, Heidelberg, 2010. Springer-Verlag.

[36] I. Fratric. Runtime prevention of return-oriented programming attacks. Technical report, 2012.

[37] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDO*, 2005.

[38] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *S&P*, 2014.

[39] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *USENIX Security*, 2014.

[40] J. Gray. Why do computers stop and what can be done about it? In *SRDS*, 1985.

[41] I. Haller, E. Göktaş, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: Vtable protection without loose ends. In *ACSAC*, 2015.

[42] I. Haller, J. Yuseok, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe. TypeSan: Practical Type Confusion Detection. In *CCS*, 2016.

[43] C. Haschek. Why are free proxies free?, 2015. https://blog.haschek.at/post/fd9bc.

[44] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a highly dependable operating system. In *EDCC*, 2006.

[45] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel. Ensuring operating system kernel integrity with OSck. In *ASPLOS*, 2011.

[46] A. Kadav, M. J. Renzelmann, and M. M. Swift. Fine-grained fault tolerance using device checkpoints. In *ASPLOS*, 2013.

[47] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 272–280, New York, NY, USA, 2003. ACM.

[48] J. W. Kim and V. Prevelakis. Base line performance measurements of access controls for libraries and modules. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 356–356, Washington, DC, USA, 2006. IEEE Computer Society.

[49] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

[50] L. Koromilas, G. Vasiliadis, E. Athanasopoulos, and S. Ioannidis. GRIM: Leveraging GPUs for kernel integrity monitoring. In *RAID*, 2016.

[51] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

[52] B. Lee, C. Song, T. Kim, and W. Lee. Type casting verification: Stopping an emerging attack vector. In *USENIX Security*, 2015.

[53] H. Lee, H. Moon, D. Jang, K. Kim, J. Lee, Y. Paek, and B. B. Kang. KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object. In *USENIX Security*, 2013.

[54] A. Lenharth, V. S. Adve, and S. T. King. Recovery domains: an organizing principle for recoverable operating systems. In *ASPLOS*, 2009.

[55] R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, 2007.

[56] M. Marlinspike. sslstrip. http://www.thoughtcrime.org/software/sslstrip/.

[57] Microsoft. Enhanced Mitigation Experience Toolkit. http://www.microsoft.com/emet, 2016.

[58] H. Moon, H. Lee, J. Lee, K. Kim, Y. Paek, and B. B. Kang. Vigilare: Toward Snoop-based Kernel Integrity Monitor. In *CCS*, 2012.

[59] R. Nikolaev and G. Back. VirtuOS: an operating system with kernel virtualization. In *SOSP*, 2013.

[60] B. Niu and G. Tan. Modular Control-flow Integrity. In *PLDI*, 2014.

[61] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in linux: ten years later. In *ASPLOS*, 2011.

[62] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *USENIX Security*, 2013.

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

[64] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *DIMVA*, 2015.

[65] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot-a coprocessor-based kernel runtime integrity monitor. In *USENIX Security*, 2004.

[66] J. Pitts. The case of the modified binaries, 2014. http://www.leviathansecurity.com/blog/the-case-of-the-modified-binaries/.

[67] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium*, USENIX Security, 2003.

[68] T. Rains, M. Miller, and D. Weston. Exploitation Trends: From Potential Risk to Actual Risk. In *RSA Conference*, 2015.

[69] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*, 2017.

[70] J. Rutkowska and A. Tereshkin. Bluepilling the xen hypervisor. *Black Hat USA*, 2008.

[71] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *ACSAC*, 2016.

[72] L. Sassaman. The faithless endpoint: How Tor puts certain users at greater risk. Technical Report ESAT-COSIC 2007-003, Katholieke Universiteit Leuven, 2007.

[73] 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 *S&P*, 2015.

[74] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz. Evaluating the effectiveness of current anti-ROP defenses. In *RAID*, 2014.

[75] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, 2006.

[76] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *SOSP*, 2007.

[77] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[78] 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 *S&P*, 2013.

[79] D. J. Sorin, M. M. Martin, M. D. Hill, D. Wood, et al. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA*, 2002.

[80] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, 2016.

[81] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *TOCS*, 1985.

[82] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Membrane: Operating system support for restartable file systems. *TOS*, 2010.

[83] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *TOCS*, 2006.

[84] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *TOCS*, 2005.

[85] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal War in Memory. In *S&P*, 2013.

[86] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting memory disclosure attacks using destructive code reads. In *CCS*, 2015.

[87] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security*, 2014.

[88] V. van der Veen, D. Andriesse, E. Goktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *CCS*, 2015.

[89] V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *S&P*, 2016.

[90] D. Vogt, C. Giuffrida, H. Bos, and A. S. Tanenbaum. Lightweight memory checkpointing. In *DSN*, 2015.

[91] J. Wang, A. Stavrou, and A. K. Ghosh. HyperCheck: A Hardware-Assisted Integrity Monitor. In *RAID*, 2010.

[92] X. Wang, L. Zhang, and P. Tanofsky. Experience report: How is dynamic symbolic execution different from manual testing? a study on KLEE. In *ISSTA*, 2015.

[93] P. Winter, R. Köwer, M. Mulazzani, M. Huber, S. Schrittwieser, S. Lindskog, and E. Weippl. Spoiled Onions: Exposing Malicious Tor Exit Relays. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, 2014.

[94] E. J. Zaborowski. Malicious proxies. DEF CON 17, 2009.

[95] M. Zalewski. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl.

[96] 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 *S&P*, 2013.

[97] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.

[98] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. Safedrive: Safe and recoverable extensions using language-based techniques. In *OSDI*, 2006.