H2020-ICT-2018-2-825377

UNICORE

UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments

Horizon 2020 - Research and Innovation Framework Programme

D3.2 Security, Safety and Validation Support Definition

Due date of deliverable: 30 September 2019

Actual submission date: 30 September 2019

Start date of project	1 January 2019
Duration	36 months
Lead contractor for this deliverable	VU Amsterdam
Version	1.0
Confidentiality status	“Public”

© UNICORE Consortium 2019

Abstract

This deliverable will describe the definition of the UNICORE security and safety primitives, which allow UNICORE applications to minimize the attack and failure surface in production. This is done both proactively (using software verification techniques) and reactively (using software hardening techniques). In addition, this deliverable will report on deterministic execution support for smart contracts.

Target Audience

The target audience for this document is all project participants.

Disclaimer

This document contains material, which is the copyright of certain UNICORE consortium parties, and may not be reproduced or copied without permission. All UNICORE consortium parties have agreed to the full publication of this document. The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the UNICORE consortium as a whole, nor a certain party of the UNICORE consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

This document does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of its content.

Impressum

Full project title	UNICORE: A Common Code Base and Toolkit for Deployment of Applications to Secure and Reliable Virtual Execution Environments
Title of the workpackage	D3.2 Security, Safety and Validation Support Definition
Editor	VU Amsterdam (VUA)
Project Co-ordinator	Emil Slusanschi, UPB
Technical Manager	Felipe Huici, NEC
Copyright notice	© 2019 Participants in project UNICORE

Executive Summary

This is the first version of the UNICORE D3.2 deliverable, “Security, Safety and Validation Support Definition”.

This deliverable will describe the definition of the UNICORE security and safety primitives, which allow UNICORE applications to minimize the attack and failure surface in production. This is done both proactively (using software verification techniques) and reactively (using software hardening techniques). In addition, this deliverable will report on deterministic execution support for smart contracts.

© UNICORE Consortium 2019

Contents

 © UNICORE Consortium 2019

List of Figures

© UNICORE Consortium 2019

1 Introduction

This deliverable reports on the progress made in task T3.2 and task T3.3 in Work-package 3 (WP3). These tasks are central to support safe and secure unikernel execution within UNICORE.

T3.2 is dedicated to the design and implementation of the UNICORE security and safety primitives. Security primitives allow UNICORE applications to minimize the attack surface in production, that is minimizing and ideally eliminating opportunities for local and remote security attacks. Similarly, safety primitives allow UNICORE applications to minimize the failure surface in production, that is minimizing and ideally eliminating opportunities for crashes. Both primitives include a portfolio of techniques to safeguard unikernel execution either proactively (e.g., using software verification techniques) or reactively (e.g., using software hardening techniques).

T3.3 is dedicated to the design and implementation of a sandbox of smart contracts within a VM to prevent access to unintended data. Other than guaranteeing isolation, this mechanism should ensure deterministic execution support. In particular, the system is designed to deterministically halt execution for smart contracts that consume an excessive amount of resources.

We start by describing the definition of the security and safety UNICORE primitives. Unless otherwise stated, we consider the latter as a subcategory of the latter. That is, we do not distinguish between denial-of-service attacks and accidental denial of service. We first analyze the attack surface, describing the threat model considered for unikernel applications and the classes of vulnerabilities/attacks attackers can exploit to mount end-to-end attacks. Next, we discuss classes of techniques we will consider within UNICORE to reduce (and ideally minimize) the attack surface, emphasizing the unikernel-specific challenges and opportunities. We conclude by reporting on deterministic execution support for smart contracts.

© UNICORE Consortium 2019

2 Attack Surface Analysis

We start with the definition of the threat model (an attacker exploiting software and/or hardware vulnerabilities). Next, we explain how the vulnerabilities in the threat model can be lifted to primitives for exploitation. We conclude by discussing how such primitives can be used to mount end-to-end attacks.

2.1 Threat Model

We assume a typical cloud (virtualization) setting, with different mutually distrusting unikernel applications co-located on the same machine. We consider both remote and local exploitation scenarios. In a remote exploitation scenario, the attacker targets a victim (networked) unikernel application by sending malicious input from a remote client over the network. In a local exploitation scenario, the attacker targets a co-located victim unikernel application by interacting with the local execution environment and/or sending malicious input to the victim.

An attacker may pursue different goals, such as denial of service, stealing sensitive information, or compromising the full system. In our threat model, we generally consider any attack whose goal is to breach *confidentiality*, *integrity*, and/or *availability*. To purse such goals, an attacker can exploit both vulnerabilities in software such as buffer overflows or use-after-free and in hardware such as cache side channels or Rowhammer. In particular, we consider an advanced attacker that can exploit multiple such vulnerabilities to craft complex exploitation primitives and mount sophisticated end-to-end attacks on the victim system. Nonetheless, we do limit our threat model to systems-level attacks. Attacks based on social engineering, insider threats, or similar are out of scope.

2.2 Attack Primitives

According to our threat model, our design needs to generally provide security across all the dimensions dictated by the "CIA" model: *confidentiality*, *integrity*, *availability*. To breach confidentiality, an attacker needs to craft primitives to leak sensitive data from memory, thus operating (unauthorized) *arbitrary memory reads*. To breach integrity, an attacker needs to craft primitives to tamper with sensitive data in memory, thus operating (unauthorized) *arbitrary memory writes*. In the following, we show how attackers can craft such arbitrary memory read/write primitives by exploiting widespread software or hardware vulnerabilities and how they can be used to mount end-to-end attacks.

2.2.1 Software-based Primitives

Software-based attack primitives are typically crafted by exploiting memory error vulnerabilities in widely deployed systems software. 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

 © UNICORE Consortium 2019

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 and write primitives and gain increased privileges.

As shown in recent studies [1], 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 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).

2.2.2 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 2.1: 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 from/to unintended locations in memory.

Listing 2.1 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.

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

2.2.3 NULL pointer dereference

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 from/to unintended locations in memory.

Listing 2.2 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.

2.2.4 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 2.3: 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

[1] https://vvdveen.com/memory-errors

© UNICORE Consortium 2019

memory access may result in a buffer overrun (or underrun) and allow an attacker to read/write from/to unintended locations in memory.

Listing 2.3 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.

2.2.5 Use-after-free

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

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 from/to unintended locations in memory.

Listing 2.4 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.

2.2.6 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 2.5: 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 from unintended locations in memory.

Listing 2.5 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.

2.2.7 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 2.6: 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 from/to unintended locations in memory.

Listing 2.6 shows an example of a typical type confusion vulnerability. The C++ function **calculate** first

 © UNICORE Consortium 2019

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.

2.2.8 Hardware-based Primitives

While software-based attacks are well-known and have been explored in the past decades by security researchers, a new class of attacks are recently being applied to hardware components found in general-purpose computing platforms. Many of these components, such as CPUs and DRAM are typically found in cloud computing platforms. We describe how an attacker could mount attacks on these components to leak secret information or fully compromise the system in a cloud.

2.2.8.1 Leaking Information from Hardware

Cloud infrastructures make use of shared hardware to increase utilization. In a typical cloud server, many tenants may share the hardware resources in that server. The most prominent shared resource in these common configurations is the CPU, where the tenants share the CPU's cores or hardware threads when they are available. While tenants are provided isolation guarantees at the architectural level of abstraction, they implicitly share many of the CPU resources at the microarchitectural level. As an example, while the Memory Management Unit (MMU) of the CPU isolates accesses to physical memory between different VMs in a server, these VMs will share various caches with each other during the course of their execution.

```
1  page = allocate();
2
3  for every cacheline l in page:
4    flush l;
5
6  wait();
7
8  for every cacheline l in page:
9    before = readtime();
10   access l;
11   after = readtime();
12
13   if (after - before) > threshold:
14     report(l);
```

Listing 2.7: an example of FLUSH+RELOAD primitive

2.2.8.1.1 Cache attacks Listing 2.7 shows a FLUSH+RELOAD primitive [1] for detecting whether a victim VM has accessed a cache location. The attacker would first allocate a page in her virtual machine. The assumption here is that this page is shared by a victim VM. This can happen for example if a technology such as memory deduplication is used in an insecure manner [2]. The aim of the attacker is to find out is how this page is accessed by a victim VM. To this end, the attacker first flushes the cache lines that store this page from various levels of caches in the CPU. The attacker then waits for a certain period of time. During this time, the victim may access the page at different locations. At this point, the attacker

can time accesses to different locations in this page. If the access at a location is fast (line 13), it means that it is in the cache and the victim has accessed this location. As we will discuss later, this information can allow the discovery of sensitive information such as cryptographic keys. FLUSH+RELOAD belongs in class of primitives that are also known as cache timing side channels. Other notable examples from this class include PRIME+PROBE [3] and EVICT+TIME [4] which assume different requirements for the attacker or the victim. For example, PRIME+PROBE does not require page sharing with the victim, but it is generally more noisy.

```
1
2  page = allocate();
3  pointer = NULL;
4
5  for every cacheline l in page:
6    flush l;
7
8  begin_transaction();
9  page[pointer];
10 end_transaction();
11
12 for every cacheline l in page:
13   before = readtime();
14   access l;
15   after = readtime();
16
17 if (after - before) > threshold:
18     report(l);
```

Listing 2.8: an example of the recent RIDL primitive

2.2.8.1.2 Speculative attacks

Cloud servers make use of powerful out of order CPUs. To execute instructions efficiently, these CPUs sometimes need to speculatively execute instructions while not knowing beforehand whether these instructions should be executed. At the later point, if the CPU realizes that these instructions should not have been executed, it will roll back those instructions and the program will not see the result of these incorrectly-executed instructions. Unfortunately, these instructions leave a trace on the microarchitectural resources (e.g., CPU caches) and this trace can be used by the attackers to leak sensitive information from the system. Listing 2.8 shows an example of the recent RIDL primitive [5]. This primitive uses the previously-discussed FLUSH+RELOAD attack to leak recent reads/writes performed by a CPU core. With RIDL, dereferencing an invalid memory address (NULL in this case) will speculatively use the recent data that is read/written by the CPU core. Hence, if we use the invalid pointer as an offset inside a page that is used for FLUSH+RELOAD (line 9), the location within the page that is cached tells us a value that the CPU has recently read or written.

2.2.8.2 Corrupting Data with Faulty Hardware

Another component that is used in cloud servers is DRAM where ephemeral data is stored. DRAM is made out of transistors which are used internally to build capacitors. To store bits of information in DRAM, these capacitors may be charged. To read back the bits, DRAM consults these capacitors to see whether they were previously charged. Capacitors lose information over time, hence, the DRAM hardware periodically refreshes these capacitors to make sure that the retain their charge over time. Similar to how the Moore's

© UNICORE Consortium 2019

law has allowed CPUs to become more efficient, it has also allowed more transistors to be placed inside DRAM chips, increasing the number of the capacitors and hence their capacity. This trend resulted in two physical properties: first, the capacitors are now much smaller and as a result they require much less charge. Second, these capacitors are now much closer to each other than previously. These new properties potentially introduce some reliability problems that can become relevant for security. Given that the DRAM hardware is implicitly shared across many users of the cloud, these security problems can affect unikernels running in the cloud.

```
1  aggressor_page1 = allocate();
2  aggressor_page2 = allocate();
3
4  victim_page = allocate();
5
6  store_sensitive_info(victim_page);
7
8  for 2 million iterations:
9
10   flush(aggressor_page1);
11   flush(aggressor_page2);
12
13   *aggressor_page1;
14   *aggressor_page2;
```

Listing 2.9: an example of a Rowhammer primitive

Listing 2.9 shows an example of a DRAM reliability problem that plagues cloud servers. Known as the Rowhammer [6] vulnerability, an attacker can access some memory locations repeatedly in quick succession to trigger bit flips in other memory locations. In this example, the attacker is accessing two memory locations (known as aggressors) to trigger a bit flip in a victim memory location that has security sensitive information (e.g., page tables). As we will discuss, Rowhammer enables powerful attacks that can compromise cloud, mobile, and browsers [7, 8, 9, 10, 11, 12, 13, **?**, 14].

2.3 Exploitation

In the following, we show how attackers can use arbitrary memory read/write primitives for exploitation purposes and briefly mention recent work in the area.

2.3.1 Software-based Exploitation

Much recent work shows how attackers can exploit arbitrary memory read/write primitives to mount end-to-end attacks and compromise a victim software. For example, an attacker-controlled memory write tampering with data that affects the control flow of a target application (i.e., *control data*) allows an attacker to mount a control-flow hijacking attack. This can be used to divert normal control flow and grant an attacker arbitrary code execution capabilities. These can, for 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, control-flow hijacking attacks based on arbitrary memory read/write primitives pose the highest security threat. And despite much recent work in the area focusing on control-flow hijacking defenses,

recent work shows that an attacker armed with such primitives can typically mount advanced control-flow hijacking attacks against even on protected systems [15, 16, 17].

Other powerful attacks based on arbitrary memory read/write primitives are also possible, depending on the setting. For example, other than relying on a *control-flow diversion* attack, attackers can opt, when possible (i.e., when the code of the victim is writeable), for a *code corruption attack* to execute arbitrary code. In a code corruption attack, attackers uses an arbitrary memory write primitive to corrupt executable code memory and inject their own malicious code. In an *information disclosure attack*, attackers use an arbitrary memory read primitive to leak control data (e.g., a function pointer from the heap) or non-control data (e.g., a stack canary) from the application. In a *data-only attack*, finally, attackers use an arbitrary memory write primitive to corrupt non-control data and gain increased privileges (e.g., a privileged user ID).

2.3.2 Hardware-based Exploitation

2.3.2.1 Cache attacks

Cache attacks are increasingly being used to leak sensitive information from a victim software component (e.g., process) running on commodity CPUs [18, 2, 19, 20, 4, 3, 1, 21]. These attacks learn about the secret operations of a victim component by observing changes in the state of various CPU caches. Since such attacks exploit fundamental hardware properties (i.e., caching), commodity software operating on security-sensitive data is inherently vulnerable. These attacks show that cryptographic keys, as well as, information about the memory layout that is important for software exploitation can easily be leaked across various security domains that are relevant for UNICORE.

2.3.2.2 Speculative execution attacks

The original Spectre attacks [22] allow attackers to manipulate the state of the branch prediction unit and abuse the mispredicted branch to leak arbitrary data within the accessible address space via a side channel (e.g., cache). This primitive by itself is useful in sandbox (e.g., JavaScript) escape scenarios, but needs to resort to *confused-deputy* attacks [23] to implement cross-address space (e.g., kernel) memory disclosure. Meltdown [24] somewhat loosens the restrictions of the addresses reachable from attacker-controlled code. Rather than restricting the code to valid addresses, an unprivileged attacker can also access privileged address space mappings that are normally made inaccessible by the supervisor bit in the translation data structures. This enables user-to-kernel attacks in a traditional user/kernel unified address space design.

Foreshadow [25] further loosens the addressing restrictions. Rather than restricting attacker-controlled code to valid and privileged addresses, the attacker can also access physical addresses mapped by invalid (e.g., non-present) translation data structure entries. Similar to Meltdown, the target physical address is accessed via the cache, data is then passed to out-of-order execution, and subsequently leaked before the corresponding invalid page fault is detected. The last of these attacks, RIDL [5], completely removes the attacker's addressing restriction and shows that it is possible to leak information without requiring a valid address. This allows RIDL to mount powerful cross-address space speculative execution attacks directly from error-free

© UNICORE Consortium 2019

and branchless unprivileged execution (including JavaScript in the browser).

2.3.2.3 Rowhammer

After its initial discovery [6] Rowhammer has been demonstrated in a plethora of attacks compromising all sorts of targets [13, 9, 8, 10, 12]. Seaborn and Dullien [26] originally demonstrated the security implications of Rowhammer compromising the Linux kernel without relying on any software bug. Afterwards, other studies proved its effectiveness to break cloud isolation [13, 27], "root" mobile platforms [9, 11] and, to make it all the more scarier, even escape sandboxed environment such as JavaScript [28, 8, 12] and trigger bit flips over the network [10]. All these attacks demonstrated the severity of the threat and showed the necessity to build effective defenses. Attacks on clouds are specially relevant in the context of UNICORE.

3 Attack Surface Reduction

In the last chapter, we reviewed the major classes of hardware and software vulnerabilities affecting unikernel applications in our threat model and showed how attackers can exploit them to mount end-to-end attacks in real-world settings. We now show how different classes of defense techniques, which we plan to consider within UNICORE, can iteratively reduce the attack surface in deployed unikernel applications. The idea is to model a deployed unikernel system as a blackbox, originally containing an unspecified number of software and hardware vulnerabilities. Our goal is to eliminate such vulnerabilities or prevent them from being successfully exploitable after deployment in production. As mentioned in D2.1, unikernel applications are relatively small and hence have a smaller attack surface compared to regular applications. Nonetheless, studies show that even mature production software averages between 1 and 16 bugs per 1,000 lines of code [29], resulting in a relevant attack surface for even small unikernel applications.

To reduce the attack surface, we consider a *multilevel* security-enhancing approach, namely a portfolio of defense techniques deployed at different stages of the system lifecycle. The first step is to deploy validation techniques that can prove the absence of vulnerabilities in software. Since such techniques cannot scale to the complexity of arbitrary unikernel applications (and remain limited to specific software modules), there are many residual vulnerabilities in software after the validation step. To reduce the number of residual vulnerabilities, the second step is to deploy fuzzing techniques and operate high-coverage security testing of the target unikernel application. Since fuzzing techniques cannot find all the possible vulnerabilities, the number of residual vulnerabilities is reduced but still relevant after the fuzzing step. To prevent the exploitation of the remaining vulnerabilities, the third step is to deploy hardening techniques for the target unikernel application. Finally, in a fourth step, we consider additional hardening techniques that protect the target unikernel application against hardware vulnerabilities. The following subsections discuss the individual classes of proposed techniques.

3.1 Validation

In this section we overview validation techniques offering security guarantees to software components followed by our approach in UNICORE. The goal is to ensure the absence of bugs whenever possible. Key to this is increasing coverage of code paths in software components; a given input results in a code path being traversed in the control flow graph of the software component and output being generated. Ideally one is able to provide the complete set of inputs that traverse all program code paths. Additionally one wants to handle concurrency, where two or more code paths are traversed simultaneously.

Approaches to validation/verification of software components are proactive. The aim is to ensure a level of correctness and security for a software component before deploying it. This is in contrast to a reactive approach, that aims to detect and limit unexpected and possibly malicious behavior of the application. As a proactive approach, validation generally requires access to source code; binary machine code can still be

© UNICORE Consortium 2019

handled, though some techniques (such as annotating source code) aren't available.

In UNICORE we target the core libraries of the Unikraft unikernel for validation. With its small code code (and reduced attack surface), Unikraft is an ideal target.

3.1.1 Overview of Validation Techniques

Verification techniques target hardware or of software components [30]. Hardware components are easier to verify than software components due to their fixed wirings and the availability of a specification. Software targets, on the other hand, are variable and complex and often lack a specification to use as support for verification.

On the software side there are three high-level practical approaches to software verification [31]: model checking, domain-specific languages and Hoare-style verification. In model checking we create a model of the program, we define properties and then we check if the model satisfies the properties. Domain-specific languages are programming languages that ensure programmers only use safe / correct features. Hoare-style verification requires the programmer to annotate specifications to source code; then it discharges the specifications and source code to a verification tool; specifications are usually written in Hoare logic [32].

Model checking verifies that a model of the software component satisfies given properties. We model a software components as a finite state machine (i.e. a graph, a control flow graph), with all reachable states for a given system must satisfy corresponding properties. Model checking ensures a given property holds for all program states. Its main disadvantage is path explosion: model checking aims to make sure a given property holds for all program states, but they can be practically infinite due to program loops. The path explosion / infinite loop issue can be tackled by bounded model checking [33]. Bounded model checking (BMC) provides a partial verification of program states expecting that it would be enough for some loops; that is it unwinds loops to a predetermined depth that may provide coverage for the entire loop (for a large enough depth). However, there are no guarantees after the given depth.

Domain-specific languages provide the programmer with a feature set preventing software components from misbehaving. Either the programmer uses only safe / correct programming constructs when writing code, or the compiler / interpreter checks the program at build time. The common case are functional programming languages that don't provide arbitrary memory access and reduce side effects. However, imperative programming languages are heavily used [34] and the transition to a domain-specific one is difficult; moreover, imperative programming languages typically provide better performance [35]. A middle ground approach is the use of imperative programming languages with security / safety features as part of the language constructs, as is the case with Rust [36] or D [37].

Hoare logic annotations have the advantage of being modular: one need not create specifications for the entire program (as is the case with model checking) but rather for a given block of code, typically a function. Annotations generally consist of a set of preconditions and set of postconditions that are fed together with source code to a verification program. The verification program translates the program to a set of predicates

and logic rules that are checked against preconditions and postconditions. One disadvantage of this approach is the effort required in creating annotations for each block of code needing to be verified and the lack of a holistic validation of the software component.

Table 3.1 summarizes the above high-level approaches, their advantages and disadvantages.

Technique	Advantage	Disadvantage
Model checking	Complete validation	Path explosion problem
Domain-specific languages	Focus on development	Commonly non-imperative languages
Hoare-style verification	Modular verification	Extensive effort on drawing annotations

Table 3.1: Summary of practical high-level approaches to validation

One technique employed for software verification is symbolic execution. Symbolic execution is an offline symbolic traversal of the control flow graph; during the traversal it checks properties for each node. Symbolic execution engines [38] are used either by themselves or as part of other approaches; model checkers typically employ symbolic execution engines.

Approaches in software validation generally require adding security properties and checking them against software components. Security properties are usually defined formally considering security policies for a given software component. Security policies might concern access control, information flow or availability [39]. Properties are defined to ensure program behaves correctly and satisfies security policies. Failure to satisfy a security property is a security error in the source code [40], one which is reported and then required to be fixed thus increasing the level of security for the software component.

3.1.2 UNICORE and Software Validation

In UNICORE we target the Unikraft unikernel for software validation. Its reduced code (and attack surface) make it an ideal target for validation. Moreover, the unikernel runs entirely isolated in its private address space, so there is little interference with the outside environment, thus simplifying the context for validation. Unikernels are a collection of libraries; this componentization of the unikernel allows the opportunity for modular validation in an iterative manner.

Based on the above, our aim to maximize security guarantees for the unikernel relies on a three-way approach:

(i) We use existing state-of-the-art techniques and approaches for verifying the unikernel: formal verification, symbolic execution, abstract interpretation, model checking. The aim is to increase coverage of unikernel code paths using existing techniques. Outcome is a framework for unikernel verification that outputs the level of guarantees for the unikernel covered code.

(ii) We use security features in modern imperative programming languages such as Rust or D. We will port unikernel components to these languages and augment them with runtime security checks (or other approaches) for select parts of these components that are not covered (or are only partially covered) by the language. Outcome are unikernel components implemented in a programming language with

 © UNICORE Consortium 2019

security features with hardened parts where security support is missing.

(iii) We define properties (specifications) for unikernel components (separately or as code annotations) and rewrite components to maximize property compliance. Outcome are unikernel components satisfying given properties.

With their reduced code base and attack surface, implicit isolation and componentization, unikernels are an ideally target for security validation. Unikraft (as part of the UNICORE project) is our target for validation. We rely on a combined approach that uses and extends state of the art approaches in software validation. In the end we will strengthen the unikernel and provide a level of security guarantees, ensuring applications running as part of the unikernel are more secure than their counterparts running as part of a general purpose operating system.

3.2 Fuzzing

Non-trivial unikernel modules are not amenable to the software validation techniques discussed in the previous section, but can still contain exploitable software vulnerabilities. With fuzzing, we attempt to eliminate such vulnerabilities and further reduce the attack surface. Fuzzing techniques run the target unikernel through a variety of carefully crafted inputs in an attempt to maximize execution coverage and find bugs. Once bugs are found, they can, in principle, be fixed during development even before the unikernel is deployed in production.

Traditional fuzzing techniques are either blackbox or whitebox. Blackbox fuzzing techniques completely ignore the internal structure of the target program and select a stream of random inputs for testing purposes. While simple and efficient, blackbox fuzzing techniques struggle to find complex bugs that lay deep in the target program execution. Whitebox fuzzing techniques, on the other hand, use systematic program analysis such as symbolic execution to thoroughly explore the internal structure of the target program. While potentially more powerful, whitebox fuzzers have trouble scaling to non-trivial programs and their practical application has remained limited. More recently, greybox fuzzers have received much attention, approximating the scalability of blackbox fuzzers and the ability to operate deep program exploration of whitebox fuzzers [41, 42]. Greybox fuzzers typically start from a blackbox fuzzing baseline (sending random inputs to the target program) and progressively mutate the input to improve testing coverage. This is done by using program instrumentation to analyze the impact of each input on the program (e.g., determining whether the input covers new program paths) and guiding the mutation based on the most promising inputs and their modifications.

Within UNICORE, we plan to build on the expertise of the consortium in greybox fuzzing and investigate efficient software fuzzing techniques for the targeted unikernel environments. Of particular interest is modeling fuzzing as a resource management problem in cloud environments. That is, determining how to best exploit bounded spare resources in production (e.g., idle times in cloud workloads) and support a continuous

testing strategy to identify residual vulnerabilities.

3.3 Software

Once residual bugs make their way to production, they may allow attackers to exploit the corresponding memory error vulnerabilities in deployed production software. To further reduce the attack surface, our next step is to counter any attempt to exploit such vulnerabilities by deploying program hardening techniques. Such defenses techniques can broadly be classified in two main categories: memory error defenses and exploit mitigations.

Memory error defenses can prevent or detect exploitation attempts for each class of memory error vulnerabilities (e.g., buffer overflow, use-after-free, uninitialized read, etc.) and stop exploits from the get-go [43, 44, 45, 46, 47, 48, 49]. While such techniques are effective at countering attacks, they cannot generally stop 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 for the target unikernel application. Hence, other solutions are also needed.

Exploit mitigations seek to eliminate the residual attack surface for memory error vulnerabilities. Once an attacker is able to trigger a memory error vulnerability (escaping deployed memory error defenses), such defenses hinder the ability to mount a successful exploit. Exploit mitigations 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 [50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60].

Within UNICORE, we plan to build on the expertise of the consortium in both classes of defenses and investigate program hardening techniques for unikernel environments. Other than the basic security features mentioned in D2.1 (absent in other unikernels), of particular interest is using hardware virtualization features available in cloud environments to accelerate defense techniques. For example, support for PML (Page Modification Logging) could be used to accelerate memory checkpointing (improving safety) and support for EPTs (Extended Page Tables) could be used to accelerate page protection-based memory error detection (improving security).

3.4 Mitigating Hardware Attacks

We discuss some of the existing defenses against hardware-based attacks.

3.4.1 Cache defenses

The security community has developed several defenses both in software and in hardware to mitigate cache side-channel attacks. Given the knowledge of how memory is mapped to the CPU caches, these defenses can freely partition the memory between distrusting processes in a way that partitions the cache, thus preventing the eviction of each other's cache lines. There are three common approaches for achieving this goal: partitioning the cache by sets [61, 62], partitioning the cache by ways [63, 64], and locking cache lines such that

© UNICORE Consortium 2019

they cannot be evicted [65, 66]. In UNICORE, we are exploring novel techniques for providing mitigation against these attacks with a focus on cloud environments.

3.4.2 Speculative execution defenses

In response to the plethora speculative execution attacks, hardware and software vendors have been struggling to catch up with mitigations that can safeguard vulnerable systems. These mitigations can operate at three different layers:

(i) inhibiting the trigger of the speculation (e.g., Retpoline [67]),

(ii) protecting the secret the attacker is trying to disclose (e.g., KPTI [68]), or

(iii) disrupting the channel of the leakage (e.g., disabling high-precision timers in browsers [69]).

In UNICORE, we are exploring scalable mitigations against speculative execution attacks that are not yet fully mitigated (such as Spectre variant 1 [22]).

3.4.3 Rowhammer defenses

Kim et al. [6] propose multiple defenses against Rowhammer. These defenses have proven insufficient [70, 71] and infeasible to deploy on the required massive scale. The new LPDDR4 standard [72] specifies two features which together defend against Rowhammer: TRR and MAC. As part of UNICORE, we plan to assess whether these new mitigations are effective against Rowhammer.

On the software side, Aweke et al. [70] suggested relying on hardware performance counters to detect suspicious access patterns that may suggest hammering activity and sending extra memory accesses from software to stop the bit flips. Instead of fixing the flips, mitigations such as CATT [73] and GuardION [11] tackle the exploitation of such bit flips and try to prevent an attacker from compromising sensitive data. Despite all these efforts, new attacks have shown how these mitigations become completely ineffective when slightly shifting the attacker paradigm [74, 12]. As a result, they are currently not considered as viable solutions. The recent ZebRAM work [75] shows that principled mitigation of Rowhammer in software is possible using Rowhammer-aware memory allocation, but at potentially non-trivial performance cost. In UNICORE, we are exploring whether it is possible to build a secure solutions with a modest performance budget in cloud environments.

4 Reference Application

After analyzing the attack surface of unikernel applications and discussing techniques to reduce it, we now to turn our attention to the reference security application for UNICORE: smart contracts.

4.1 Smart Contracts

A smart contract is a computer program that runs in a distributed infrastructure and whose results are certified or verified in a distributed manner. Smart contracts computation typically uses blockchain technology. Smart contracts do not require a centralized entity but rely on cryptographic primitives implemented by the decentralized blockchain infrastructure to be validated.

The successful deployment of a smart contract in a distributed infrastructure relies on nodes reaching consensus, i.e. multiple nodes in the infrastructure reaching the same results when running the smart contract. Ambiguities, bugs, environment configuration may lead to different results for different nodes, intentionally or non-intentionally, breaking consensus and tampering the certification of the smart contract. As such there is a requirement for deterministic execution of a smart contract, i.e. given the same input, the smart contract will output the same result irrespective of the running environment: hardware/software platform, resource use, configuration, timing.

Validating the smart contract and the underlying infrastructure for determinism is paramount to its successful run.

Current approaches to ensure safety in smart contracts [76] rely on a domain-specific language and a dedicated infrastructure. One of the most popular infrastructures is EVM (*Ethereum Virtual Machine*). EVM is the infrastructure used by the Ethereum platform [77] to run smart contracts; smart contracts are written in Solidity [78], a highly constrained language that ensures safety for smart contracts.

In UNICORE we want to use the unikernel technology to develop smart contracts in high-level popular imperative languages such as Go, or Python, or even C. This will increase the number of smart contracts developers and make it easier to develop smart contracts. This approach relies on the Unikraft unikernel to provide the support for deterministic execution: isolated environment, reduced code base, validation of software components.

Deterministic execution for smart contract programs running on top of unikernel libraries depends on validation of the unikernel. With the unikernel satisfying correctness properties, the smart contract can add further steps in ensuring deterministic execution: architectural compatibility, disabling scheduling, threading and timing, controlling input/output to the program.

In summary, there is a two-fold effort in ensuring deterministic execution for smart contracts written in high-level languages running within a unikernel:

(i) Validate/Harden the unikernel. The aim is to eliminate / minimize memory corruption issues, control-flow hijack attacks, misuse of data, information leaks.

 © UNICORE Consortium 2019

(ii) Control smart contract source code, build options and running configuration. The aim is to ensure deterministic execution of smart contracts in all configurations, i.e. given the same input but differing environments, the output and other effects are identical.

5 Conclusions

This deliverable described the definition of the security and safety UNICORE primitives as well as the deterministic execution support for smart contracts. To define the security and safety UNICORE primitives, we considered a typical could (virtualization) threat model with local and remote attacks. From the threat model, we analyzed the corresponding attack surface, discussing the relevant exploitation primitives and techniques attackers may use to breach confidentiality, integrity, and availability of unikernel applications. Overall, we showed unikernels are exposed to several different threats, ranging from exploitation based on software vulnerabilities to side channels and fault attacks.

Next, we discussed classes of defensive techniques that can be used to reduce the attack surface. Proactive techniques such as formal verification, when applicable, can structurally eliminate specific threats such as memory error exploitation. Reactive techniques such as isolation or program hardening can help reduce (and ideally eliminate) the attack surface. Overall, we showed unikernels are amenable to a broad variety of techniques to reduce the attack surface. In UNICORE, we will consider different classes of such techniques applied at different layers in the stack to support safe and secure unikernel execution. We concluded the deliverable by reporting on our progress in supporting deterministic execution for smart contracts.

© UNICORE Consortium 2019

References

[1] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*, (San Diego, CA), pp. 719–732, USENIX Association, Aug. 2014.

[2] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure Page Fusion with VUsion," in *SOSP*, Oct. 2017.

[3] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015.

[4] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pp. 1–20, 2006.

[5] S. van Schaik, A. Milburn, S. Osterlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-flight Data Load," in *S&P*, May 2019. Intel Bounty Reward, Pwnie Award Nomination for Most Innovative Research.

[6] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 361–372, IEEE Press, 2014.

[7] A. Tatar, C. Giuffrida, H. Bos, and K. Razavi, "Defeating Software Mitigations against Rowhammer: A Surgical Precision Hammer," in *RAID*, Sept. 2018. Best Paper Award.

[8] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida, "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector," in *S&P*, May 2016. Pwnie Award for Most Innovative Research.

[9] V. van der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida, "Drammer: Deterministic Rowhammer Attacks on Mobile Platforms," in *CCS*, Oct. 2016. Pwnie Award for Best Privilege Escalation Bug, Android Security Reward, CSAW Best Paper Award, DCSR Paper Award.

[10] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *USENIX ATC*, July 2018. Pwnie Award Nomination for Most Innovative Research.

[11] V. van der Veen, M. Lindorfer, Y. Fratantonio, H. Padmanabha Pillai, G. Vigna, C. Kruegel, H. Bos, and K. Razavi, "GuardION: Practical Mitigation of DMA-based Rowhammer Attacks on ARM," in *DIMVA*, June 2018. Pwnie Award Nomination for Best Privilege Escalation Bug.

[12] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi, "Grand Pwning Unit: Accelerating Microarchitectural Attacks with the GPU," in *S&P*, May 2018. Pwnie Award Nomination for Most Innovative Research.

[13] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos, "Flip Feng Shui: Rowhammering the VM's Isolation," in *Black Hat Europe*, Nov. 2016.

[14] S. Hong, P. Frigo, Y. Kaya, C. Giuffrida, and T. Dumitras, "Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks Under Hardware Fault Attacks," in *USENIX Security*, Aug. 2019.

[15] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida, "The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later," in *CCS*, Oct. 2017.

[16] E. Goktas, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos, "Undermining Information Hiding (And What to do About it)," in *USENIX Security*, Aug. 2016.

[17] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida, "Poking Holes in Information Hiding," in *USENIX Security*, Aug. 2016.

[18] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "ASLR on the Line: Practical Cache Attacks on the MMU," in *NDSS*, Feb. 2017. Pwnie Award for Most Innovative Research, DCSR Paper Award.

[19] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks Against Kernel Space ASLR," S&P '13.

[20] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," CCS '16.

[21] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security*, Aug. 2018. Pwnie Award Nomination for Most Innovative Research.

[22] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P'19*.

[23] S. van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder Than You Think," in *USENIX Security*, Aug. 2018.

 © UNICORE Consortium 2019

[24] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security'18*.

[25] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security'18*.

[26] M. Seaborn and T. Dullien, "Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges," in *BHUS'15*, 2015.

[27] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation.," in *USENIX Security'16*, 2016.

[28] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA'16* (J. Caballero, U. Zurutuza, and R. J. Rodríguez, eds.), Lecture Notes in Computer Science, pp. 300–321, Springer International Publishing, 2016.

[29] 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*, June 2016. Best Paper Session.

[30] O. Demir, W. Xiong, F. Zaghloul, and J. Szefer, "Survey of Approaches for Security Verification of Hardware/Software Systems," *IACR Cryptology ePrint Archive*, vol. 2016, p. 846, 2016.

[31] Y. Song, A. Costea, and W.-N. Chin, "Automated modular verification for continuous time systems via a temporal effects logic," *Unpublished manuscript*, 2019.

[32] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, Oct. 1969.

[33] Edmund, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, pp. 7–34, Jul 2001.

[34] "PYPL PopularitY of Programming Language." http://pypl.github.io/PYPL.html, 2019. [Online; accessed 25-September-2019].

[35] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. a. P. Fernandes, and J. a. Saraiva, "Energy efficiency across programming languages: How do energy, time, and memory relate?," in *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, (New York, NY, USA), pp. 256–267, ACM, 2017.

[36] "Rust." https://www.rust-lang.org, 2019. [Online; accessed 25-September-2019].

© UNICORE Consortium 2019

[37] "The D language." https://dlang.org, 2019. [Online; accessed 25-September-2019].

[38] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, pp. 50:1–50:39, May 2018.

[39] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, pp. 30–50, Feb. 2000.

[40] K. Tsipenyuk, B. Chess, and G. McGraw, "Seven pernicious kingdoms: a taxonomy of software security errors," *IEEE Security Privacy*, vol. 3, pp. 81–84, Nov 2005.

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

[42] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "TIFF: Using Input Type Inference To Improve Fuzzing," in *ACSAC*, Dec. 2018.

[43] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta Pointers: Buffer Overflow Checks Without the Checks," in *EuroSys*, Apr. 2018.

[44] T. Kroes, K. Koning, C. Giuffrida, H. Bos, and E. van der Kouwe, "Fast and Generic Metadata Management with Mid-Fat Pointers," in *EuroSec*, Apr. 2017.

[45] I. Haller, E. van der Kouwe, C. Giuffrida, and H. Bos, "METAlloc: Efficient and Comprehensive Metadata Management for Software Security Hardening," in *EuroSec*, Apr. 2016.

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

[47] E. van der Kouwe, V. Nigade, and C. Giuffrida, "DangSan: Scalable Use-after-free Detection," in *EuroSys*, Apr. 2017.

[48] A. Milburn, H. Bos, and C. Giuffrida, "SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities," in *NDSS*, Feb. 2017.

[49] E. van der Kouwe, T. Kroes, C. Ouwehand, H. Bos, and C. Giuffrida, "Type-After-Type: Practical and Complete Type-Safe Memory Reuse," in *ACSAC*, Dec. 2018.

[50] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, and H. Bos, "ShrinkWrap: VTable Protection Without Loose Ends," in *ACSAC*, Oct. 2015. Outstanding Student Paper Award.

[51] K. Koning, H. Bos, and C. Giuffrida, "Secure and Efficient Multi-variant Execution Using Hardware-assisted Process Virtualization," in *DSN*, June 2016.

 © UNICORE Consortium 2019

[52] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, “VTPin: Practical VTable Hijacking Protection for Binaries,” in *ACSAC*, Dec. 2016.

[53] 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*, May 2016.

[54] X. Chen, H. Bos, and C. Giuffrida, “CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks,” in *EuroS&P*, Apr. 2017.

[55] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida, “MARX: Uncovering Class Hierarchies in C++ Programs,” in *NDSS*, Feb. 2017.

[56] 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*, Nov. 2015.

[57] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida, “StackArmor: Comprehensive Protection from Stack-based Memory Error Vulnerabilities for Binaries,” in *NDSS*, Nov. 2015.

[58] A. Slowinska, T. Stancescu, and H. Bos, “Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation,” in *USENIX ATC*, Nov. 2012.

[59] S. Osterlund, K. Koning, P. Olivier, A. Barbalace, H. Bos, and C. Giuffrida, “kMVX: Detecting Kernel Information Leaks with Multi-variant Execution,” in *ASPLOS*, Apr. 2019.

[60] A. Pawlowski, V. van der Veen, D. Andriesse, E. van der Kouwe, T. Holz, C. Giuffrida, and H. Bos, “VPS: Excavating High-Level C++ Constructs from Low-Level Binaries to Protect Dynamic Dispatching,” in *ACSAC*, Dec. 2019.

[61] Y. Ye, R. West, Z. Cheng, and Y. Li, “COLORIS: A Dynamic Cache Partitioning System Using Page Coloring,” PACT ’14.

[62] Z. Zhou, M. K. Reiter, and Y. Zhang, “A Software Approach to Defeating Side Channels in Last-Level Caches,” CCS ’16.

[63] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, “CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing,” HPCA ’16.

[64] C. Intel, “Improving Real-Time Performance by Utilizing Cache Allocation Technology,” *Intel Corporation, April*, 2015.

[65] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, “Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory,” USENIX Security ’17.

© UNICORE Consortium 2019

[66] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud," USENIX Security '12.

[67] P. Turner, "Retpoline: a Software Construct for Preventing Branch Target Injection." https://support.google.com/faqs/answer/7625886, Jan 2018.

[68] "KPTI - Linux Documentation." https://www.kernel.org/doc/Documentation/x86/pti.txt Retrieved 15.10.2018.

[69] M. Bynens, "Untrusted Code Mitigations." https://v8.dev/docs/untrusted-code-mitigations, Jan 2018.

[70] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-Based Protection Against Next-Generation Rowhammer Attacks," ASPLOS'16.

[71] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *S&P*, May 2019. Best Practical Paper Award, Pwnie Award Nomination for Most Innovative Research.

[72] JEDEC Solid State Technology Association, "Low Power Double Data 4 (LPDDR4)," *JESD209-4A*, 2015.

[73] F. Brasser, L. Davi, D. Gens, C. Liebchen, and A.-R. Sadeghi, "CAn't Touch This: Practical and Generic Software-only Defenses Against Rowhammer Attacks," *arXiv preprint arXiv:1611.08396*, 2016.

[74] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, , and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," *arXiv preprint arXiv:1710.00551*, 2017.

[75] R. K. Konoth, M. Oliverio, A. Tatar, D. Andriesse, H. Bos, C. Giuffrida, and K. Razavi, "ZebRAM: Comprehensive and Compatible Software Protection Against Rowhammer Attacks," in *OSDI*, Oct. 2018.

[76] D. Harz and W. J. Knottenbelt, "Towards safer smart contracts: A survey of languages and verification methods," *CoRR*, vol. abs/1809.09805, 2018.

[77] "Ethereum." https://www.ethereum.org, 2019. [Online; accessed 25-September-2019].

[78] "Solidity." https://github.com/ethereum/solidity, 2019. [Online; accessed 25-September-2019].

 © UNICORE Consortium 2019