European Commission - Horizon 2020 DS-07-2017
Cybersecurity PPP: Addressing Advanced Cyber Security Threats and Threat Actors

REactively Defending against Advanced Cybersecurity Threats [†]

WP2: Attack Scenarios
Deliverable D2.3: Technology Requirements

Abstract:

Security defenses are applied transparently on systems for mitigating attacks. These defenses follow different strategies. As a first example, consider the approach of analyzing systems for discovering vulnerabilities; as a second example, consider the approach of enhancing a program with additional code for making exploitation much harder. *ReAct* incorporates a large arsenal of different approaches for reactively protecting systems. No matter the particular approach, all methods interact with or modify existing systems. Therefore, it is crucial to map (a) the technologies that *ReAct* uses for delivering protection, and (b) the platforms that *ReAct* can handle. We call all these *technology requirements*. In this deliverable, we list all technical requirements for the three core work-packages, namely WP3, WP4, and WP5.

Contractual Date of Upload	May 2019
Actual Date of Upload	May 2019
Deliverable Dissemination Level	Public
Editor	Elias Athanasopoulos
Contributors	All *ReAct* partners
Quality Assurance	Herbert Bos

[†] This project is funded by the European Commission (Horizon 2020 - DS-07-2017) under Grant agreement no: 786669.

The *ReAct* Consortium

FORTH	Coordinator	Greece
STICHTING VU	Beneficiary	The Netherlands
UNIVERSITY OF CYPRUS	Beneficiary	Cyprus
EURECOM	Beneficiary	France
RUHR-UNIVERSITAET BOCHUM	Beneficiary	Germany
SYMANTEC	Beneficiary	France

Document Revisions & Quality Assurance

Internal Reviewers

1. Herbert Bos
2. Cristiano Giuffrida
3. Davide Balzarotti

Revisions

Ver.	Date	By	Overview
1.0.3	30/5/2019	*Elias Athanasopoulos*	Addressed comments of internal review.
1.0.2	24/5/2019	*Elias Athanasopoulos*	Addressed comments of coordinator. New draft for QMC.
1.0.1	22/5/2019	*Elias Athanasopoulos*	Pre-submission draft.
1.0.0	30/4/2019	*Elias Athanasopoulos*	First draft.

Contents

List of Figures

1 Introduction

Computer security can affect several systems, nowadays, making the term *computer* quite vague. Systems are not uniform anymore but rather diverse. Essentially, there are several *different* but in parallel *similar* computers in our lives, from fitness recorders that contain a fully functional computer on our wrist to software engines that power modern vehicles. All these systems can be affected by security attacks; nevertheless, their differences, in terms of their underline technology, produce variations in both attacking and defending these devices.

> **Attack Scenario.** In *ReAct* we define as an *attack scenario* a list of assumptions about a specific attack targeting a very well-defined system. Part of an attack scenario is the *threat model*, which defines the capabilities of an attacker. In D2.1, we carried out a broad assessment of current security trends, and we concluded with the interesting threat models for *ReAct*.

In this deliverable, D2.3, we investigate the technologies used for countering the threats identified in D2.1. Producing security defenses for defined threat models can be fairly complicated when several different types of devices come into play. Especially, for frameworks such as *ReAct*, this is particularly interesting. First, *ReAct* aims at countering threats in different states of activity. As an example, *ReAct* applies different techniques to *vulnerable* or *exploited* hosts. Second, for each state, different types of technologies are operational. For instance, an *ordinary* host will be scanned using fuzzing for vulnerabilities, and an identified *vulnerable* host will be selectively fortified by means of software hardening. These two technologies, fuzzing and software hardening, have different foundations and realizing them requires a set of different techniques and technologies. Moreover, we stress here that

fuzzing and software hardening are used only as examples, since *ReAct* uses *additional* technologies, depending on the type of host.

Beyond these two, namely that different techniques apply to host depending of its state and that different techniques utilize diverse underline technologies, there is a third *important* issue. Most of the *ReAct* technologies incorporate system-level defenses, which are aligned with very precise technologies, and *receivers* (i.e., systems that are protected by *ReAct*) need to support certain features in order to get benefit from the framework. This will become clear later in this deliverable, where we expand on how our techniques work and how they get benefit from existing technologies, broadly used in the market.

> **Technology Requirements.** We define as technology requirements the set of system properties that need to be in place for realizing certain techniques of *ReAct*. For example, a *ReAct* technique may target software of a particular computer architecture or it may depend on a very specific CPU feature. The set of these properties compose the *technology requirements* of *ReAct*.

Last but not least, this deliverable also highlights systems that *ReAct* cannot protect. Notice that *ReAct* incorporates methodologies that are *fairly* generic. In that sense, *ReAct* is designed to work with a broad range of systems, nevertheless, implementing the techniques for every exotic architecture out there can be challenging. Therefore, although it is clear that certain systems are not, currently, supported, our techniques can still stand as a *contribution*, since, they are likely to be applied to additional systems in the future.

Dashboard. This deliverable does not contain any information about technologies required for realizing the user interface of *ReAct*. All these activities are part of WP6, and they are documented in deliverables of WP6. This deliverable is related only to the technologies required for implementing the core techniques of *ReAct*, which are activities of WP3, WP4, and WP5.

1.1 Executive Summary

This deliverable presents all techniques that are planned to be realized as part of the activities of WP3, WP4, and WP5. Each technique is presented in high level and then a list of technology dependencies per technique is given. Some key methodologies that cover several individual techniques are the following.

- **Software Instrumentation.** The ability to enhance a program with additional functionality is vital for several techniques of *ReAct*. Some examples are: (a) analyzing software for vulnerabilities through gray-box fuzzing, (b) selectively hardening a vulnerable part of a program, (c) adding functionality to existing software for a future forensics analysis. In all these cases, *ReAct* transforms software either by working with Intel binaries or LLVM bitcode (a summary is later given in Table 5.1).
- **Code monitoring.** Several techniques of *ReAct* need access to the execution history of a running program. Some examples are: (a) monitoring the instructions triggered by a supplied input during a fuzzing session, (b) detecting malicious probes. For these cases, *ReAct* leverages the Intel Processor Trace, which is available in all recent Intel CPUs (a summary is later given in Table 5.1).

1.2 Organization

This deliverable is organized as follows. In Chapter 2, 3, and 4, we briefly refresh the reader with the systems of *ReAct* per each technical WP. We keep the content minimal, and we emphasize only the core components that have particular technological dependencies. Later on, in Chapter 5 we summarize all dependencies and we conclude with Chapter 6.

2

WP3: Data Collection and Target Identification

ReAct aims at proactively identifying vulnerabilities and forecasting future threats. These are core components of the framework and allow us to be one step ahead of the attackers. Identification of vulnerabilities and forecasting of future threats are realized using different techniques, and each one has their set of technology dependencies. In this part, we discuss some preliminaries of how vulnerabilities can be found and how data of a host may signal future attacks. Then, we summarize the technology dependencies of WP3.

2.1 Finding Unknown Vulnerabilities

Once a vulnerability is known, the affected system can be analyzed and patched. The process of finding vulnerabilities in software can be fairly challenging and complicated [9]. Nevertheless, finding mechanically vulnerabilities is important, since it allows the defenders to apply the right fixes before an attack takes place. We stress that both defenders and attackers try to find new vulnerabilities in software for different reasons. Attackers need to find new vulnerabilities for developing exploits and attacking systems, while defenders need to find the vulnerabilities first, for producing patches, and contain exploit attempts.

For *ReAct*, finding vulnerabilities in an *automated* fashion is a central concept, since it allows us to bootstrap complementary techniques that patch software automatically [2, 3] (these techniques are discussed in Chapter 3). We will expand on all details of the techniques that identify software vulnerabilities using computer programs in future deliverables of WP3. Here, we just discuss the high-level idea for helping the reader to understand why certain system features must be present. The core concept of our methodology for automatically finding software vulnerabilities is broadly named as *fuzzing* [6, 13, 12].

2.1.1 Fuzzing Software

ReAct, as discussed in Deliverable 2.1, is interested in software exploitation through memory-corruption vulnerabilities. This type of errors is triggered by sending a malicious input to the victim program. It is tempting to explore if those *malicious inputs* can be discovered using a computer algorithm.

One critical observation is that, for a given bug, there might be several malicious inputs that can stimulate it. For example, consider a malicious input that, if sent, forces a program to download malware, or, another malicious input, that, if sent, forces a program to open a remote connection to the attacker. Both malicious inputs trigger the same vulnerability, but, since they have different mechanics, the inputs are different, as well. A second critical observation is that a specific bug may be triggered through different code paths. For instance, a function that has a buffer overflow may be called by multiple call sites. A third critical observation is that a bug may allow the attacker to corrupt different data. For example, a stack vulnerability may be used to modify a return address, other control data (e.g., a function or vtable pointer), or other non-control, but sensitive, data. Finally, a fourth critical observation is that, if the input is not accurately computed then the program may crash. Since, these inputs trigger memory corruption, then just corrupting memory will make the program crash.

This is very important, since we can compute random inputs, send them to the program, and inspect if any of those random inputs forces the program to crash. If yes, then we can thoroughly analyze the input and see if the analyzed software, while processing the given input, executes code that contains a memory-corruption vulnerability. This approach, sending auto-generated inputs to an analyzed program, and inspect if any of those inputs crashes the program is called *fuzzing* [15].

Now, from a first read, fuzzing looks like a straightforward, not really complicated, technique. A researcher creates a program that computes random inputs, sends it to an under-analysis program and inspects if any of those inputs make the program crash. Although this might be intuitive, such a naive technique is unlikely to produce any meaningful results, since programs are fairly complicated and the input space is huge. Randomly and blindly computing inputs is not the most efficient approach. For instance, if we could, somehow, get some feedback from the under-analysis program, then the input computation could be significantly improved. For example, if several inputs stimulate the exact same code path, without producing any crash, then it might be a bad investment to produce additional inputs that explore once again this particular code path. A better strategy would be to compute inputs that explore *different* code paths.

```
/* magic number example */
if(u64(input)==u64("MAGICHDR"))
    bug(1);

/* nested checksum example */
if (u64(input)==sum(input+8, len-8))
    if (u64(input+8)==sum(input+16, len-16))
        if (input[16]=='R' && input[17]=='Q')
            bug(2);
```

Figure 2.1: Challenges in fuzzing code. The first bug can only be found if the first 8 bytes of the input are a *specific* magic header. To reach the second bug, the input has to contain the string "RQ" and two correct checksums. The probability of randomly creating an input that satisfies these conditions is negligible.

2.1.2 Fuzzing Challenges

§In order to understand the need for the *feedback loop*, we discuss here some very common problems that appear when fuzzing even fairly simple programs. These problems are broadly known as *magic numbers* and *checksum tests*. An example for such code can be seen in Figure 2.1. Assume that a program, which is being fuzzed, contains the depicted code. Also, assume that the under-analysis program contains two vulnerabilities, depicted in the code as `bug(1)` and `bug(2)`.

Now, let's further assume that the variable *input* is sent to the program using a fuzzing tool. Each time, the fuzzer selects a new *random* (or perturbed) input and inspects the program for a crash. The first bug can only be found if the first 8 bytes of the input are a *specific* magic header. To reach the second bug, the input has to contain the string "RQ" and two correct checksums. The probability of randomly creating an input that satisfies these conditions is negligible. Therefore, a simple approach does not produce new coverage, the fuzzing process stalls, and the technique is not productive.

This example demonstrates two things: (a) the need for a feedback loop, for carefully selecting a new input, since there are cases where several inputs will not make any progress for the fuzzer, and (b) this feedback loop must be carefully implemented, otherwise there is high probability that the feedback is not useful enough for producing a *good* new input. We stress that there is always the trade-off between producing good inputs and generating inputs rapidly. Sometimes generating a new input as fast as possible is best, sometimes it is better to spend a bit more time computing about what could be a good input.

2.1.3 Implementing the Feedback Loop

So far, we have argued about the importance of fuzzing in finding new vulnerabilities and especially we have discussed how a feedback loop may assist the fuzzer to reach a useful input much quicker than blindly selecting random ones. Clearly, there are several different strategies for realizing an efficient feedback loop. Nonetheless, however the feedback loop decides the information for sending it back to the fuzzer, and however the fuzzer utilizes this data, there should be some underlying decisions that should be taken. These decisions are the following.

- How is the analyzed software instrumented?
- What is the underlying platform?
- How should we monitor which code has been executed?

We expand on these, here.

How is the analyzed software instrumented?

The feedback loop needs access to the software of the program. For instance, the feedback loop may discover magic numbers, branches, or other constructs (see Figure 2.1) for helping the fuzzer to progress. Now, this software instrumentation can be done either at the binary level, or at the source level. In the first case, when the feedback loop instruments the binary, the underlying architecture is important (for instance, instrumenting an Intel binary is different than instrumenting an ARM one). In the second case, there might be better portability, but, again, the level where the instrumentation is applied is important. A common approach is to do all instrumentation at the LLVM level [7]. LLVM is a very common intermediate representation (IR), used by several compiler front-ends. Therefore a C or C++ (or Objective-C) compiler may produce LLVM bitcode, which at a later stage, can be further compiled to machine code. Since the LLVM format is broadly known, and several tools exist for manipulating, several fuzzers chose to implement all software instrumentation at this level. In *ReAct*, our fuzzing techniques will involve LLVM instrumentation, when source is available.

What is the underlying platform?

The feedback loop is implemented by instrumenting software, therefore, the underlying architecture can play an important role. In cases where source code is available, instrumentation can be done in a more portable fashion. As already discussed above, several compilers support LLVM, a popular intermediate representation form. In such cases, fuzzing can be agnostic as

far as the underlying architecture is concerned. However, there are cases where source is not available and software must be instrumented at the binary level. In such cases, the underlying architecture is very tightly connected with the binary representation. For *ReAct*, the techniques related, focus only on the Intel architecture (both 32 and 64 bit), while several of those methodologies could be ported in other popular architectures, such as ARM.

The underlying architecture can also play a role when software, through instrumentation, is analyzed for inferring the inputs that will lead to new code paths. In these cases, software should be analyzed in terms of the actual code executing, such as tight loops or taken branches. These code constructs are implemented differently in each underlying architecture.

How is the analyzed software instrumented?

Beyond software instrumentation, which is vital for realizing the feedback loop, it is also important to have access to instructions that have been already executed. This gives the instrumentation the ability to infer how inputs interact with the fuzzed software. Recording software execution for analyzing it at a later stage of a fuzzing session can increase the effectiveness of the feedback loop. In *ReAct* we take advantage of modern recording features available in recent Intel CPUs, such as Intel Processor Trace [5].

2.2 Forecasting Future Threats

A critical component of *ReAct* is entirely pro-active and it is based on pinpointing future attack targets. This is based on threat modelling and on analyzing a variety of data, which may include system's details, as well as user habits and procedures. Compared to the rest of the techniques, this pro-active component is not bound to specific technologies, since modelling a system can include a broad set of system properties. Therefore, techniques for forecasting future threats do not have specific technology requirements.

2.3 Summary of Technology Dependencies

To conclude this part, all activities of WP3 have as a major target the Intel architecture, namely x86 and x86-64. This architecture contains the vast majority of running desktop and laptop computers, today. Although the 32-bit version (x86) is now rather obsolete, it is not uncommon that 32-bit software can run on 64-bit machines (x86-64). Additionally, some of the techniques of WP3 utilize advanced CPU debugging features, such as Intel Processor Trace, and some virtualization options appearing only in recent Intel CPUs. As far as software is concerned, WP3 can work with binaries

Table 2.1: All activities of WP3 have as a major target the Intel architecture, namely x86 and x86-64. They require the presence of Intel Processor Trace, and some virtualization options (Intel VT-x). As far as software is concerned, WP3 can work with binaries produced by a C/C++ compiler or with source (LLVM).

Technique	**Architecture**	**Operating System**	**Software**	**Hardware**	**Other**
Fuzzing	x86, x86-64	Linux, Microsoft Windows	Binaries, LLVM	Intel Processor Trace, Intel VT-x	-
Forecasting	Not applicable	Linux, Microsoft Windows, OSX	-	-	-

produced by a C/C++ compiler or with source (LLVM). We summarize all these dependencies in Table 2.1.

3 WP4: Real-time Patching

Once a vulnerable or compromised host is discovered, *ReAct* needs to take action. For vulnerable hosts, *ReAct* produces *automated* patches, by means of software instrumentation/hardening, and, for compromised hosts, *ReAct* isolates them from the rest of the network. In this chapter, we discuss which technologies are needed for selectively fortifying a vulnerable host and for isolating a compromised host.

3.1 Selective Hardening

Producing a *software patch* in an automated fashion is fairly challenging. *ReAct* does this for two core reasons: (a) as we have thoroughly discussed in D2.1, one of the critical reasons that software can be exploited today is the reluctance of people in applying patches, (b) a completely fortified program, i.e., a program that has employed all known (overly conservative) mitigations and cannot be exploited, introduces an unacceptable overhead [14, 11, 10]. Therefore, we need to address both (a) and (b). That is, we need to assume that a program has exploitable vulnerabilities, but, once these are found, then the program can be automatically patched, without the introduction of high overheads.

3.1.1 Software hardening

Unsafe systems produce binaries that run with very little run-time support (such as garbage collection, bounds checking, etc.) and therefore these programs can access their memory in a totally unrestricted way. In several cases this is very useful and fast; for instance, low-level software takes advantage of this loose memory-accessing model. However, there are some drawbacks. Essentially, the consequences of unsafe code having software vulnerabilities can be severe, since memory corruption can be leveraged for completely

controlling the vulnerable program. A possible solution is to re-write unsafe programs to programming systems that enforce safety, however, this can result in (a) very slow software, (b) heavily constrained software (some programs *need* unrestricted access to memory). Last but not least, this type of re-writing code is considered complex.

An alternative solution that seems to close the gap between totally unsafe programs and very constrained safe programs is *software hardening* of unsafe code [1, 16, 4]. These techniques can selectively add protections on top of unsafe programs. The goal of software hardening is to add only the necessary protections so that unsafe programs receive safety without sacrificing significantly their performance. These protections come usually in the form of *checks* embedded in the program's code. For example, whenever a *return address* is read from the stack of the program, it is checked for inferring if the value of this address has been corrupted; if so, the corruption indicates a possible exploitation attempt.

Inferring if the value of control data stored in the memory of the running process has been maliciously modified or not can be realized with several techniques. We will expand on the precise techniques that *ReAct* uses for software hardening in future deliverables. Currently, the crucial parts that are of interest are the following:

1. How do we put (additional) checks in a program?

2. What is the cost of putting additional checks in a program?

Answering (1) depends on the underlying architecture. In *ReAct* we have a twofold approach. Primarily, we aim at instrumenting LLVM bitcode. This is a fairly portable technique, since several compilers produce LLVM bitcode as an intermediate-representation form. Instrumenting LLVM bitcode can happen by extending a compiler that supports LLVM or by reading (and enhancing) the produced LLVM bitcode. In both cases, it is clear that the instrumented software needs re-compilation and, therefore, the source code of the program should be present. In cases, where the source code is not available, *ReAct* can apply binary-only techniques, however, these can be only applied to x86 and x86-64 binaries.

For answering (2), it is clear that additional checks produce overhead. An unsafe program can receive a vast amount of checks, which will essentially transform it to a *safe* program, however the performance penalty might be unrealistic. *ReAct* takes a novel approach to this trade-off. Initially, the running program is softly instrumented — essentially, the program is *lightly annotated* for receiving *heavy* instrumentation at a later stage. Once a vulnerability is discovered, then *ReAct* adds heavy instrumentation *only* to the vulnerable part of the program. Beyond discovering new vulnerabilities, *ReAct* also keeps monitoring the running program for any occurrence of

probing attempts using a set of "anomaly" detectors. When the *ReAct* defense encounters any such attempt, again, it automatically locates its origin, and patches only the offending piece of code at runtime with stronger and more expensive integrity-based defenses. The real mechanics of this process will be discussed in a future deliverable. At this point, we elaborate only in the necessary technologies that must be present. For realizing this type of *selective* software hardening we need to annotate LLVM bitcode (or x86/x86-64 binaries), and apply heavy instrumentation once a vulnerability (or exploitation attempt) is found.

3.1.2 Program Monitoring

As and when necessary, *ReAct* applies heavy instrumentation on limited parts of executing software and this can be considered as an automated, temporary, patch. For applying this instrumentation, *ReAct* needs to acquire the details of a vulnerability *or* infer an active exploitation attempt. Thus, it is important for *ReAct* to have access to traces of past execution. Like in fuzzing, which is discussed in Chapter 2, *ReAct* employs the advanced recording feature available in modern Intel CPUs for this, namely Intel Processor Trace [5].

3.2 Isolation

In this section we focus on isolating hosts that have been detected to be infected. There are two steps to this process:

- Host Detection
- Host Isolation

3.2.1 Host Detection

Detecting that a host is compromised may sound easy: we just need to run an anti-virus software on the host and based on the results of the antivirus we conclude whether the host is compromised or not. After all, this is what we do everyday with our personal computers, our desktops and our laptops.

Unfortunately, the situation is a bit more complicated. The main difficulty here is that we may not have (physical or virtual) access to the compromised host. For example, we may not have an account on the host, or we may not have the permissions to run any software (including an antivirus). There exist even cases where adding software on a host may void the guarantee. As a result it is very common than not that administrators do not have (physical or virtual) access to the compromised hosts. In these cases detection should be done from a distance: this is sometimes called

remote detection. In this project we use a two-pronged approach for remote detection:

- Active Monitoring
- Passive Monitoring

3.2.1.1 Active Monitoring

In this approach we *probe* the remote hosts for any signs of vulnerability or compromization. We *probe* (i) for open ports and (ii) for running software (that "listens" to these ports) that is vulnerable.

3.2.1.2 Passive Monitoring

This approach is a bit more complicated. It assumes that we have access to the network traffic sent or received by the compromised host. It assumes that we are able to monitor this traffic, and scan for any issues of vulnerability, compromization or attack. [1] For example, the WannaCry [8] attack is done via network packets that contain the following payload:

```
|FF|SMB3|00 00 00 00|
```

So, if we detect a host that sends this payload to other computers[2], it means that the host is compromized and is attacking other hosts.

3.2.2 Host Isolation

One a computer is detected to be infected some action needs to be taken. One possible action would be to *isolate* the computer. That is, *detach* the computer from the rest of the Internet so that it can do no harm to other computers. *Isolating* a host might sound simple: we just walk onto the computer, turn it off, and we are done. After all, don't we do the same thing when our laptop/desktop freezes or misbehaves?

Unfortunately, it is a bit more complicated. The main problem is again that we may not have (physical) access to the infected computer. Indeed,

[1] This assumption, although not unrealistic, is not always true. The main reason is that access to a host's traffic is a bit challenging. It usually requires the collaboration of the host (to be scanned) or the gateway router.

[2] Actually, it is a bit more complicated than this. The `|FF|SMB3|00 00 00 00|` payload has to be in particular positions (depth) of an established TCP connection. Thus, packets need to be re-assembled into data streams including out-of-order and overlapping packets. Once the data stream is re-assembled the particular pattern (i.e. `|FF|SMB3|00 00 00 00|`) has to be present at a given depth in the data stream. Furthermore, this established TCP connection should be directed from a host inside the local network towards another host - most usually in an outside network. For the benefit of simplicity we omit such detailed description in the text.

the computer may be behind a locked door, may be geographically far away, or may not have a protocol that will safely shut it down and not cause any harm to its data or its users. In these cases, we use the help of network routers to isolate the infected computer. In particular, we assume that we have access to network routers and can issue commands that will block all traffic of infected computers. [3] Essentially we provide network routers with the IP address of the infected computer, along with an instruction to block all traffic to (or from) this computer. After the block instruction is executed, all traffic of the infected computer will not be able to reach the outside world. The computer will observe that "the Internet is down" and it will not be able to communicate with computers beyond the router.

3.3 Summary of Technology Dependencies

3.3.1 Technology Dependencies for Selective Fortification

For selective fortification the major target is the Intel architecture, namely x86 and x86-64. This architecture contains the vast majority of running desktop and laptop computers, today. Although the 32-bit version (x86) is now rather obsolete, it is not uncommon that 32-bit software can run on 64-bit machines (x86-64). Additionally, some of the techniques for selective fortification utilize advanced CPU debugging features, such as Intel Processor Trace. As far as software is concerned, selective fortification can work by instrumenting source code (LLVM IR). We summarize all this in dependencies in Table 3.1.

3.3.2 Technology Dependencies for Compromised Host Detection and Isolation

- *Access to Network Traffic:* To be able to detect that a host is compromised and is attacking other hosts we need access to its network traffic.
- *Root Access to Gateway Router:* To be able to isolate the compromised host, that is, to block all traffic to/from the host we need to be able to have root access to its gateway router and issue commands that will block all traffic to/from the compromized host.
- *IP address:* We assume that the compromised hosts has its own IP address which is not shared with other computers. This is because the

[3] For the purpose of discussion we assume that we block all traffic of the infected computer. However, we have the ability to *manipulate* the infected computer's traffic in more sophisticated ways. For example, we may *delay* traffic, *throttle* traffic, block traffic only from/to specific sources/destinations, etc.

Table 3.1: All activities of WP4 have as a major target the Intel architecture, namely x86 and x86-64. They require the presence of Intel Processor Trace and software that can be compiled to LLVM.

Technique	**Architecture**	**Operating System**	**Software**	**Hardware**	**Other**
Real-time Patching	x86, x86-64	Linux, Microsoft Windows	LLVM	Intel Processor Trace	-
Detection	-	-	SNORT (or similar)	Access to Network traffic (unencrypted)	
Isolation	-	Linux, Microsoft Windows, OSX	-	Root Access to Gateway Router of the infected host	The compromized host has non-shareable IP address

"unit" of routing (and isolation) in the IP protocol is the IP address. If the host "shares" its IP address with other computers, then isolating one of them will effectively isolate them all.

4

WP5: Forensics Readiness

There are cases where *ReAct* is unable to protect a compromised host. If this happens, the host is detected and isolated with techniques discussed in WP4. Additionally, the host can be further analyzed for assessing the severity of the incident and infer potentially carried-out malicious activities. This phase is covered by the forensics services of *ReAct*.

In fact, there are two major components:

- **Software Instrumentation.** Inferring what happened, once a host is compromised is a highly tedious and hard task. However, software can help the analyst if it is a priori prepared to collect and store rich information. Unfortunately, most of the existing software today is not instrumented for collecting data, which is useful for forensic investigation. In WP5 we apply software instrumentation for preparing existing programs to collect data that can help us upon a security incident.

- **Data Analysis.** Collected data need to be parsed and analyzed for extracting meaningful results. Data will be processed by custom tools, that will be developed in WP5.

Technology requirements are needed only for the software-instrumentation part. In data analysis we use our own tools for processing all the collected information.

4.1 Instrumenting Software for Forensics Analysis

A forensics investigation is carried out upon a security incident has already happened. The investigation analyzes data stored in the system for inferring the malicious actions that were executed. The whole procedure can become quite involved, since malicious actions may attempt to make the forensics

investigation fairly hard (for instance, by creating fake data or hiding the actual actions).

Additionally, software is not designed for a future forensics analysis. Programs are developed to expose a certain functionality; it is unlikely that this functionality includes collecting not strictly necessary data. Sometimes, software comes in debugging mode, which allows for a more verbose logging, nevertheless, in most cases: (a) debugging information is usually turned off, for performance reasons, and (b) even when available, debugging information is not necessarily helpful for a forensics investigation.

In *ReAct*, we acknowledge that hosts may be compromised and we provide a novel approach for improving the results of a forensics investigation. We instrument selective software, which qualifies as an important target, and we enhance it with additional functionality, which is important for the forensics analyst. We stress here, that the input software has not been programmed for collecting more information than needed for its ordinary functionality. It is our techniques that come on top of existing software and transform it to *forensics-ready*.

4.2 Selective Instrumentation

Instrumenting a program for collecting useful data for a possible forensics analysis must be done with extreme caution. First, consider that a forensics analysis is a rare event, since it will take place only in the unfortunate case of a serious system compromise. Second, instrumentation adds functionality which does not come for free, since extra code usually results to performance degradation. Third, not all parts of a program are associated with the creation and collection of data that is interesting from a forensics perspective. Therefore, it is vital that all instrumentation targeting forensics readiness is applied to software selectively. The process of determining which parts of a program need to receive instrumentation for a forensics analysis will be analyzed in future deliverables of WP5.

4.3 Software Type

Software instrumentation, especially when applied to generic software, can be challenging. Recall that we leverage software instrumentation in other techniques of WP3 and WP4. In particular, in WP3 we instrument the analyzed program to give useful feedback back to the fuzzer and, in WP4, we use software instrumentation for realizing selective fortification (similar to applying temporary software patches). The instrumentation we employ in WP5 is for transforming a forensics-agnostic program to a forensics-ready one. For this, we will investigate several options to apply our modifications,

Table 4.1: WP5 applies instrumentation for programs running on Linux and the Intel architecture. Instrumentation is applied at the LLVM level.

Technique	**Architecture**	**Operating System**	**Software**	**Hardware**	**Other**
Forensics	x86, x86-64	Linux	LLVM, Binaries	-	-

including LLVM passes, binary rewriting, and other system-based instrumentations based on hardware features.

4.4 Summary of Technology Dependencies

We summarize the dependencies for forensics readiness in Table 4.1.

5 Technology Requirements

We now summarize all technologies required for the techniques realized in WP3, WP4, and WP5. These dependencies are related to the following activities (per WP):

- **WP3**. Discovering vulnerabilities through fuzzing and forecasting future targets,
- **WP4**. Selectively hardening vulnerable parts of a program and isolating compromised hosts.
- **WP5**. Prepare programs for forensics analysis.

Dependencies are summarized in Table 5.1.

Table 5.1: Summary of all the technology dependencies for the activities of WP3, WP4, and WP5.

Technique	**Architecture**	**Operating System**	**Software**	**Hardware**	**Other**
WP3					
Fuzzing	x86, x86-64	Linux, Microsoft Windows	Binaries, LLVM	Intel Processor Trace, Intel VT-x	-
Forecasting	Not applicable	Linux, Microsoft Windows, OSX	-	-	-
WP4					
Real-time Patching	x86, x86-64	Linux, Microsoft Windows	LLVM	Intel Processor Trace	-
Detection	-	-	SNORT (or similar)	Access to Network traffic (unencrypted)	
Isolation	-	Linux, Microsoft Windows, OSX	-	Root Access to Gateway Router of the infected host	The compromized host has non-shareable IP address
WP4					
Forensics	x86, x86-64	Linux	LLVM, Binaries	-	-

6 Conclusion

In this deliverable we list all technologies required for realizing the core techniques of *ReAct*, which are part of the activities in WP3, WP4, and WP5. Some of the key techniques that are based on certain technologies are the following.

- **Software Instrumentation.** The ability to enhance a program with additional functionality is vital for several techniques of *ReAct*. Some examples are: (a) analyzing software for vulnerabilities through gray-box fuzzing, (b) selectively hardening a vulnerable part of a program, (c) adding functionality to existing software for a future forensics analysis. In all these cases, *ReAct* transforms software either by working with Intel binaries or LLVM bitcode (see a summary in Table 5.1).
- **Code monitoring.** Several techniques of *ReAct* need access to the execution history of a running program. Some examples are: (a) monitoring the instructions triggered by a supplied input during a fuzzing session, (b) detecting malicious probes. For these cases, *ReAct* leverages the Intel Processor Trace, which is available in all recent Intel CPUs (see a summary in Table 5.1).

Beyond those key techniques, we have also expanded on technology dependencies by complementary methods, such as host isolation at the network level.

Last but not least, this deliverable is associated with task T2.3 which ends in M24. It is possible that new technologies may be introduced in the market, which might be suitable for optimizing the techniques of *ReAct*.

Bibliography

[1] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*. The Internet Society, 2015.

[2] C. Giuffrida, C. Iorgulescu, G. Tamburrelli, and A. S. Tanenbaum. Automating live update for generic server programs. *IEEE Trans. Software Eng.*, 43(3):207–225, 2017.

[3] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Safe and automatic live update for operating systems. In V. Sarkar and R. Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, pages 279–292. ACM, 2013.

[4] I. Haller, E. Göktas, E. Athanasopoulos, G. Portokalidis, and H. Bos. Shrinkwrap: Vtable protection without loose ends. In *ACSAC*, pages 341–350. ACM, 2015.

[5] A. Kleen and B. Strong. Intel processor trace on linux. *Tracing Summit*, 2015, 2015.

[6] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. ACM.

[7] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[8] S. Mohurle and M. Patil. A brief study of wannacry threat: Ransomware attack 2017. *International Journal of Advanced Research in Computer Science*, 8(5), 2017.

[9] D. Mu, A. Cuevas, L. Yang, H. Hu, X. Xing, B. Mao, and G. Wang. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 919–936, Baltimore, MD, 2018. USENIX Association.

[10] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 31–40, New York, NY, USA, 2010. ACM.

[11] G. Novark and E. D. Berger. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[12] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 697–710, 2018.

[13] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, 2017.

[14] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[15] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[16] 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 *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 941–955, Berkeley, CA, USA, 2014. USENIX Association.