D5.4 Appendix L

Additional test and results reported outside the Verticals scope

Project number	830892
Project acronym	SPARTA
Project title	Strategic programs for advanced research and technology in Europe
Start date of the project	1st February, 2019
Duration	36 months
Programme	H2020-SU-ICT-2018-2020

Deliverable type	Report
Deliverable reference number	SU-ICT-03-830892 / D5.4 / V1.0 / Appendix L
Work package contributing to the deliverable	WP5
Due date	Jan 2022 – M36
Actual submission date	2nd February, 2022

Responsible organisation	CNIT
Editor	Gabriele Restuccia
Dissemination level	PU
Revision	V1.0

Abstract	This document presents a list of test and results reported outside the Verticals' scope for tools that were presented as "stand-alone" or that need to be showed in an external scenario, not necessarily tied to the CAPE Verticals' use-cases.
Keywords	prototypes, certification, validation, test, result, stand-alone

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 830892.

Editor

Gabriele Restuccia (CNIT)

Contributors (ordered according to beneficiary numbers)

Henrik Plate (SAP)

Marc Ohm (UBO)

Paul-Henri Mignot, Gregory Blanc (IMT)

Gabriele Restuccia, Alessandro Pellegrini, Francesco Quaglia (CNIT)

Mirko Malacario, Claudio Porretti (LEO)

Reviewers

Maximilian Tschirschnitz (TUM)

Rimantas Zylius (L3CE)

Table of Content

List of Figures

List of Tables

Chapter 1 Introduction

This document presents a list of test and results reported outside the Verticals' scope for tools that were presented as "stand-alone" or that need to be showed in an external scenario, not necessarily tied to the CAPE Verticals' use-cases.

1.1 Subset of CAPE Tools showed

The next table shows a list representing the subset of CAPE tools which we have chosen to show outside the Verticals' scope. Note that some of them still belong to a specific Scenario of the Verticals: that is because some (or all) of their results might need to be showed by themselves. Other tools might be also "stand-alone" by design.

Tool	Partner	V-model Phase	Task	Scenario
Buildwatch (BW)	UBO	Application development	T5.3	e-Government (Vertical 2)
Legitimate Traffic Generation System (LTGen)	IMT	Operations	T5.1	Stand-alone
SATRA (ex NeSSoS)	CNR	Risk Management process at the global level	T5.1	e-Government (Vertical 2)
Steady/Project KB (KB)	SAP	All phases	T5.3	e-Government (Vertical 2)
SideChannelDefuse (FS)	CNIT	Deployment	T5.1	Stand-alone

Table 1: CAPE Tools showed outside Verticals' scope

1.2 Structure of the Document

- **Chapter** 1 is the current section presenting an Introduction and the Structure of the document
- **Chapter 2** presents the tests and results SATRA tool from CNR
- **Chapter 3** presents the tests and results for the Buildwatch tool from UBO
- **Chapter 4** presents the tests and results for the SideChannelDefuse tool from CNIT
- **Chapter 5** presents the tests and results for the Steady/ProjectKB from SAP
- **Chapter 6** presents the tests and results for the Legitimate Traffic Generation system from IMT

Chapter 2 SATRA (ex NeSSoS tool) - CNR

2.1 SATRA

SATRA is a Self-Assessment Tool for Risk Analysis (this tool evolved from the previous its version, called NeSSoS, much due to the modifications made for the SPARTA project). The tool has been adapted for the needs of SPARTA to compute cyber security risks of software products.

SATRA has the goal to provide a simple, fast and standardised way for assessment of cyber risks. This allows the tool to be useful as for the direct use (when a user interacts with the tool directly), as well as to be integrated with other tools and be able to conduct the risk analysis dynamically.

For using SATRA, two sets of inputs are required. The first set, is the knowledge about security functional requirements and security assurance practices applied for securing the software product. This knowledge is provided in a form of a questionnaire, with the concrete questions based on the basic cyber security development practices described in ISO/IEC 15408 (Common Criteria) standard.

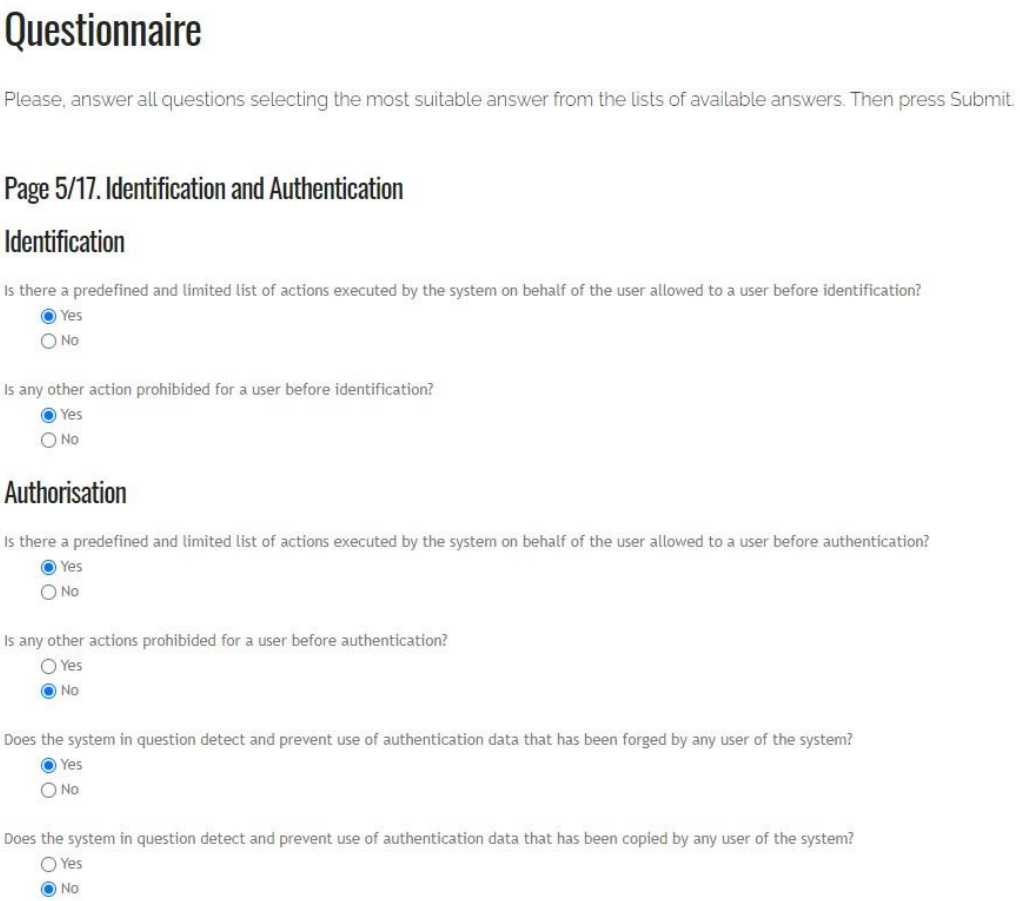

Figure 1: Questionnaire of SATRA

The second set of inputs is the list of main assets and estimated losses per product in case Confidentiality, Integrity and Availability of the software product is compromised. The main assets consist of: 1) Process, i.e., the software product itself and its internal data; 2) External process, i.e., the context in which the software product if expected to be used; 3) a set of user data, processed, transmitted or stored by the software product. The estimated losses are provided as a value between 1 to 10, assuming that 1 is negligible loss, while 10 is a catastrophically loss (e.g., leasing to loss of many human lives).

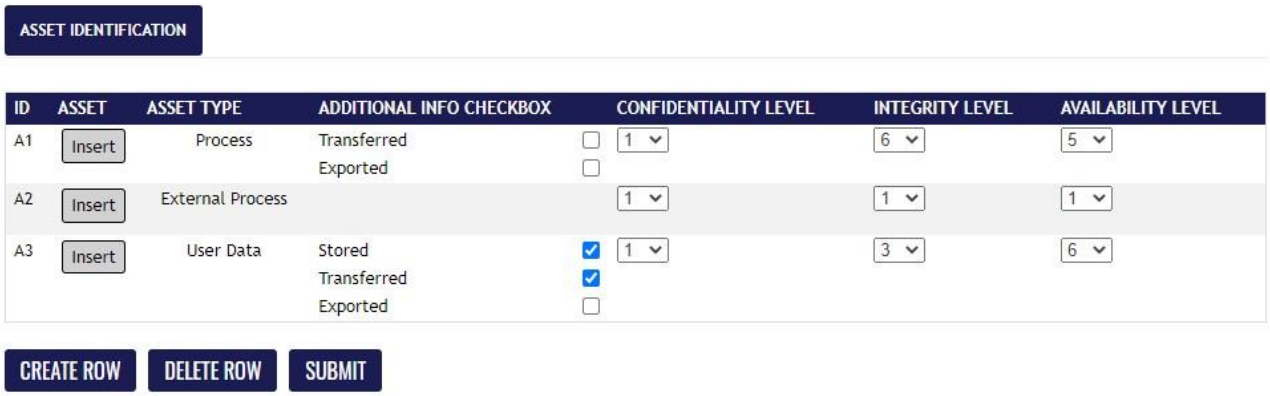

Figure 2: Asset page of SATRA

As a result, SATRA computes the overall risk value (a value from 0 to 100), and specific risk values per each of 6 STRIDE threats. The STRIDE threats were used as this methodology they are the ones among the most common for threat assessment.

In scope of Vertical 2 of SPARTA, SATRA can be used to estimate risks of both mobile APP and SAML IdP service. It could be of use at different stages and by different actors.

First, the developer may use SATRA to estimate risks for the software to be developed at the planning phase, when security requirements and strictness of the security development practices to be applied is selected. This can be required to estimate if the selected approach to securing the developed software is good enough, to identify poorly covered threats/risks, and increase protection for the relevant security aspects.

Second, the risk assessment conducted by SATRA can be used to show how secure is the software developed product. The obtained risk results could be used first to see (and prove) how much better is the final software product protected in comparison to the “unprotected” version. Secondly, it can be used to compare different versions of the same system (and see how much the risk rises if the new versions do not follow some of the initially established risk assessment practices). Last but not least, the results of the assessment can be used to compare similar products, since SATRA produces absolute (not relative) final values.

Finally, SATRA may compute results using objectively and dynamically provided (if continuous evaluation is possible) measures provided by external tools. Although possible this capability of SATRA is not used in Vertical 2 (but applied in Vertical 1).

The SATRA tool is composed of three different modules, two for the interfaces (web platform and RESTful APIs) and one for the inner-work and computations. The engine module is written in Java, while the RESTful APIs are written in Python and follow the OpenAPI version 3 standard. The development of SATRA takes place in a git environment. Each module runs in a different Docker container and the entire tool and the communications between modules is managed by docker-compose.

2.2 Integration into the CI pipeline

SATRA may be integrated into an existing repository project as a CI job. A new user of the SATRA tool (ideally, a developer of the project), has to register himself to the platform by asking for access credentials from an administrator.

In the registration phase, the user account is linked to his/her GitLab project. The newly registered user receives an email for finalizing the registration on the SATRA web platform [6].

Logged into the platform, the user finds a list of contracts, one for each GitLab project he/she has registered. The contracts have different statuses: when a contract is “Open”, it is required for the user to complete a survey regarding the security measures and information about the GitLab project under analysis.

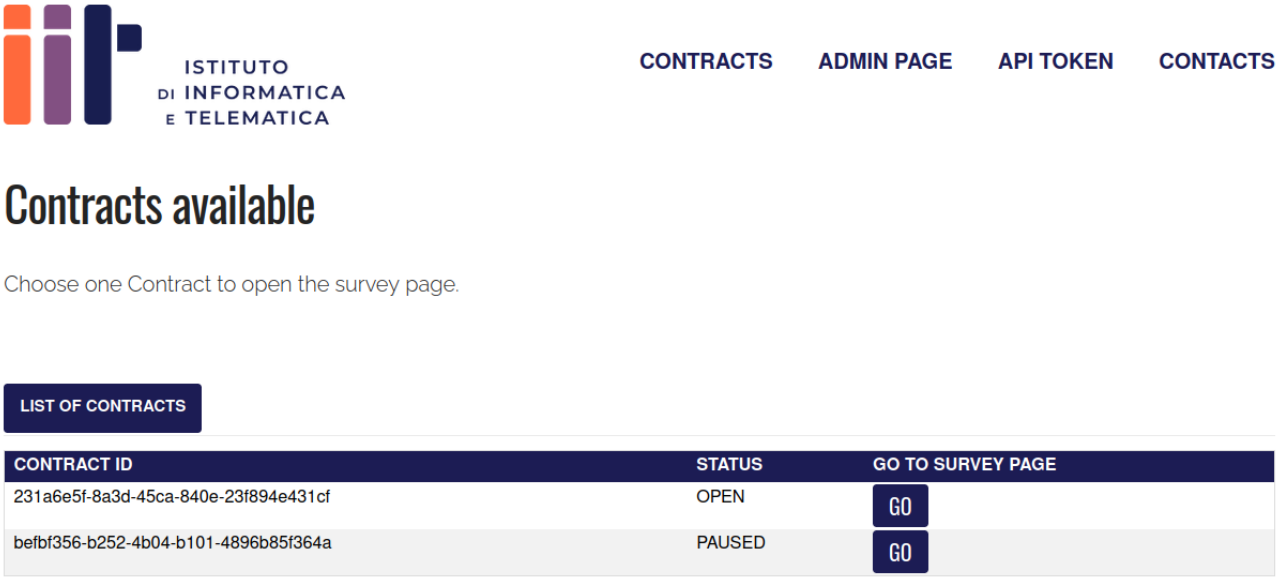

Figure 3: SATRA web page, when the user log in into the web platform

When the survey is completely fulfilled, the tool returns a risk evaluation based on the answers provided and shows the compliance percentage with regard to some Information Security categories. The survey can also be answered through the APIs [7], using the contract id (called UUID) and the personal API token provided in the web platform [8].

Figure 4: Example of the Risk Evaluation results provided when the survey is completed.

When the survey is completed, the contract changes to a “Paused” state: the information on the survey cannot be changed until a new change is committed on the GitLab repository. Moreover, a GitLab issue with the risk computation results is automatically opened on the GitLab repository when the survey is submitted.

2.3 Results evaluation

Once the analyst receives the results of the risk analysis, it is possible to see how well the tool addresses security issues. The analyst can see the level of risk, estimated by the tool, according to

all security measures used in the security development process. Now, the analyst may decide if the risk is acceptable. It is advisable to compare the results, e.g., with no-security case (to see the amount of risk reduction), with a previous version (to monitor risk changes) or with a similar system (to compare alternatives).

The analyst can also see the detailed risks per STRIDE threats. In the example in Figure 4, it is clear that tampering, information disclosure, DoS and elevation of privileges are the most important risk. This is because the associated potential losses are high for this example since confidentiality is high for the user data to be stored and processed, and integrity and availability is high for the tool itself. Nevertheless, these risks are significantly reduced by the security measures applied during the software development.

2.4 Conclusion

SATRA provides a simple and fast way to analyse risks for a software product. It can be integrated with the CI pipeline to ensure that all relevant risks are properly addressed and compare the risk levels. The results could be used by as by the developers and analysts of mobile APP as well as for the SAML IdP service for securing the system of Vertical 2.

Analysing the results, one should keep in mind that risk results will hardly be close to zero for the software products with potentially high losses (e.g., the critical ones or the ones managing large amount of user data). This is because despite following good security development practices, there could always be ways for a hacker to discover and exploit a new vulnerability (this is especially true for complex products). Thus, even if the reduction in risks is high (e.g., ten times or more), potential losses may still keep risk significant enough.

Chapter 3 Buildwatch - UBO

3.1 Presentation of the repository under test

Shibboleth is a single sign on system often used by federations like universities or other public service organizations. Shibboleth itself is written in Java leveraging the Spring framework. It can be build from source using Apache Maven.

The development of Shibboleth takes place in a git environment [9]. There, all changes are tracked and version increments are labeled accordingly. For our experiment, the component “Shibboleth Identity Provider” will be examined. This component is responsible for managing user data and corresponding authentication requests.

As Buildwatch focuses on changes introduced when a software component is updated, three consecutive versions of Shibboleth Identity Provider were chosen. In order to proof that Buildwatch is able to secure the Shibbolet as presented in D5.2 [1] the git repository was cloned and analyzed for suspicious changes in the version increments from v4.1.2 to v4.1.3 and from v4.1.3 to v.4.1.4.

3.2 Running Buildwatch as part of a CI pipeline

```
Running with gitlab-runner 14.0.1 (c1edb478)
  on buildwatch-shell Asw5D86i
Preparing the "shell" executor                                    00:00
Using Shell executor...
Preparing environment                                             00:00
Running on buildwatchaio...
Getting source from Git repository                                00:01
Fetching changes with git depth set to 50...
Reinitialized existing Git repository in /home/buildwatch/builds/Asw5D86i/0/oh
m/buildwatch-test/.git/
Checking out 81e90b64 as main...
Removing .buildwatch.sh
Removing project.zip
Skipping Git submodules setup
Executing "step_script" stage of the job script
$ echo "touch test.dat" > $CI_PROJECT_DIR/.buildwatch.sh
$ cd $CI_PROJECT_DIR && zip -r project.zip . -x '*.git*'
  adding: .buildwatch.sh (stored 0%)
  adding: do.sh (deflated 3%)
$ ID=$(curl -s -X POST -H "Content-Type: multipart/form-data" -F "file=@projec
t.zip" localhost:8080/run/uploadZip)
$ RUN=$(curl -s -X POST -H "Content-Type: application/json" -d "{\"project_id
\": \"1\", \"user_set_identifier\": \"${CI_COMMIT_SHORT_SHA}\", \"zip_filename
\": \"${ID}\"}" localhost:8080/run | jq ".id")
$ STATUS=$(curl -o /dev/null -I -L -s -w "%{http_code}" localhost:8080/run/${RU
N}/report/json)
$ while [ $STATUS -ne 200 ]; do sleep 10; STATUS=$(curl -o /dev/null -I -L -s -
w "%{http_code}" localhost:8080/run/${RUN}/report/json); done; echo "done"
done
$ curl -s localhost:8080/run/${RUN}/report/json | jq
{
  "files_removed": [],
  "files_read": [
    "/lib/x86_64-linux-gnu/libc.so.6",
    "/usr/lib/locale/locale-archive",
    "/tmpCuckoo/.buildwatch.sh",
    "/etc/ld.so.cache"
  ],
  "domains_connected": [],
  "files_written": [
    "/tmpCuckoo/test.dat"
  ],
  "hosts_connected": [],
  "processes_created": [
    "/bin/sh /tmpCuckoo/.buildwatch.sh",
    "touch test.dat",
    "sh -c /tmpCuckoo/.buildwatch.sh",
    "/tmpCuckoo/.buildwatch.sh"
  ]
}
Cleaning up file based variables                                  00:00
Job succeeded
```

Figure 5: New Commit on Buildwatch

Buildwatch may be integrated into an existing software project as a CI job. Buildwatch itself runs in a Docker container but leverages Cuckoo and VirtualBox for safe execution of the build instructions. We leveraged GitLab and the GitLab CI to trigger the analyses of the different versions of Shibboleth Identity Provider.

As seen on Figure 5 a new commit (or in our case version) will trigger the analysis. The source code and corresponding build instructions are transferred into a isolated sandbox. There the software is build according to the instructions. All forensic artifacts, like created files and connected hosts, are recorded. In the default configuration the software is build three times in a clean (reset) sandbox in order to get more reproducible artifacts.

Recorded forensic artifacts are grouped by categories, e.g. “files created”, “files removed”, “processes created”, and send back as JSON file to the CI job. In the screenshot on the left, that JSON is simply printed as part of the job’s log. Further processing of the JSON file is possible.

When a new commit is pushed or a new version is tagged, Buildwatch will automatically compare all recorded artifacts to the already known ones of previous runs. This way the focus is on newly introduced changed.

For the evaluation on Shibboleth, we used the build instructions as provided by CINI/FBK:

mvn -DskipTests -Dmaven.javadoc.disable=1 clean install

3.3 Results

For the very first analysis of an software by Buildwatch a very verbose output is to be expected. Because there are no known artifacts to remove from the report, it will include all observed artifacts. Thus, the first report generated for v4.1.2 comprised

- 13,898 files that were written
- 14,914 files that were read
- 1,912 files that were removed
- 4 hosts that were connected
- 1 domain that was contacted
- 15 processes that were created

It is not feasible to analyze all these recorded artifacts by hand. In order to reduce the number of artifacts Buildwatch tries to generate patterns of artifacts that can safely be removed the next time. For that first run as mentioned above, Buildwatch was able to generate 1206 patterns.

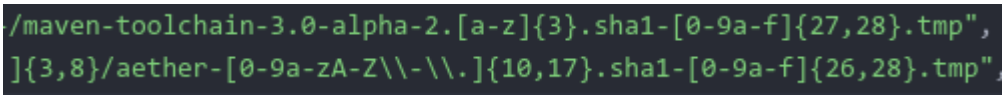

Figure 6: Example of two patterns that handle name suffixes based on the sha1 hashsum of the software components.

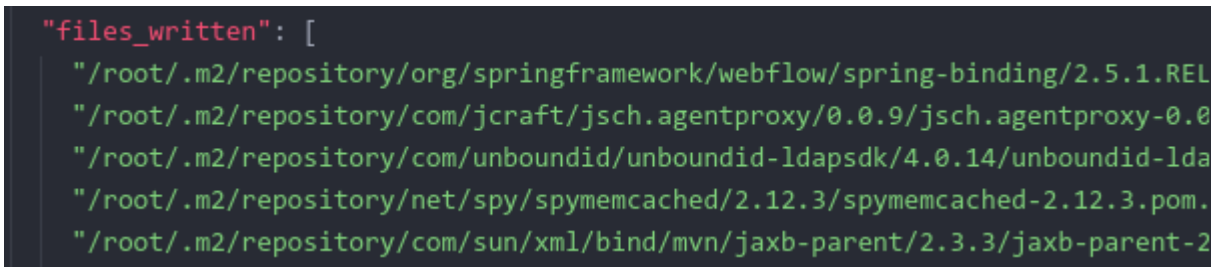

Figure 7: Small snippet of the report for v4.1.4. It is easy to see that these five third-party components were updated to the also visible version number.

For the next version (v4.1.3) the amount of presented artifacts was drastically reduced. Only 1475 new files were written, 1402 read, and 184 removed. However, no new process, host, or domain was observed. It is visible that most of the new files (written and read) are due to updated dependencies. The identification of the updated component could easily be identified from the artifacts. Based on newly observed artifacts Buildwatch was able to generate more patterns for benign artifacts, increasing the number of patterns to 1673.

In the next version of Shibboleth Identity Provider (v4.1.4), 197 new files were written and 77 were read. Again, no new hosts, domain, or processes were observed. This time even no new file deletion could be observed. Again, new patterns could be created increasing the number of patterns to 1750.

3.4 Conclusion

Shibboleth is huge projects that causes ten-thousands of forensic artifacts when build. Identifying suspicious or even malicious artifacts is like finding a needle in a haystack. Buildwatch tackles this ambitious goal by learning a software's expected behavior over the time.

As shown above, the first results are very generic as Buildwatch is not yet able to distinguish new from normal. Beginning with the second use of Buildwatch, known and benign artifacts are removed by extrapolated patterns.

For the second and thirst build no unusual network activity during the build (CWE-200) could be identified. Even though the number of recorded file operations is high, most artifacts stem from updates of used third-party components. A more detailed look at this will also ensure that there are no trojanized dependencies present in the software (TOP10-A9-2017-N1). The remaining file-related artifacts can be used to determine whether a dependency changed its expected behavior between versions (CWE-439) or that other unusual filesystem activity during the build are present (CWE-284).

In conclusion, Buildwatch is able to strengthen the security of software by enabling insight into changes in the software's behavior introduced by a developer or leveraged third-party components.

Chapter 4 SideChannelDefuse - CNIT

SideChannelDefuse is a tool for continuous kernel-level system-wide detection, assessment, and reactive mitigation against side-channel attacks. This detection is continuous, because the (host) operating system kernel-based detection mechanism is always on, while introducing a minimal overhead in the system. It is system-level, in the sense that it monitors all applications running in the system and does assume which process is the attacker and which is the victim.

If the tool detects that a (virtualized) application is trying to carry out a side-channel attack, that application is deemed as suspected. At this stage, the tool can activate per-application mitigation mechanisms, which is to reduce the likelihood that the application can exfiltrate data using the attack. The tool is developed as a patch to the Linux kernel.

The cloud owner/maintainer deploys the patched kernel into the host system and lets it run. The tool continuously runs in the background. It relies on the use of **Performance Monitoring Units (PMUs)** equipped into modern CPUs to profile the performance or (to some extent) the behavior of applications. PMUs are composed basically of programmable **Performance Monitor Counters (PMCs)** also referred to as Hardware **Performance Counters (HPCs)**.

The tool currently only targets the Intel architecture, considering its widespread nature [10] and the fact that it has been repeatedly subject to multiple attacks in the last years. Nevertheless, as we discuss, our reference implementation can be easily ported to other architectures, such as AMD.

4.1 Detecting Side-Channel Attacks

The tool relies on a combination of measures taken from HPCs in real-time, which allows us to discriminate processes that are more likely to perform operations on the cache hierarchy, indicating that they are mounting a side-channel attack. At the kernel level, four major components are involved in the system-wide monitoring of the attacks to detect the activity of malicious processes. The Monitor module directly interacts with hardware performance counters, programming them to acquire the measures to build our detection metrics. Data coming from HPCs are stored directly in a process' `task_struct`. The Detector module relies on these data to compute detection metrics and deem a running process as suspected or not—again, this information is stored in the `task_struct`. If a process is suspected, the Mitigator module will detect it and apply proper mitigations. The Scheduler module interacts with the operating system's scheduler. It is one of the fundamental components to enable system-wide detection and per-process mitigations: every time a different `task_struct` is scheduled, both the Mitigator and the Monitor modules are notified to enable/disable mitigations and reprogram HPCs, respectively, to account for the newly scheduled process.

4.1.1 Detection Metrics

Considering inclusive caching systems, which represent our target, we know that bringing the cache into a given state (e.g. a cache line is flushed or a cache set is primed) means performing an operation that is necessarily reflected into the state of caches at all the levels, from **First-Level Cache (L1)** to **Lowest-Level Cache (LLC)**. We decided to relate to different volumes of micro-architectural events generated at different levels within the caching system. First, we know that the exploitation of a side channel is based on bringing the caching system into a known initial state. Successively, the attacker attempts to determine whether some change has occurred in the cache state. At the same time, we wanted to focus on events that are not easily manipulable (in terms of their volume generation at a specific cache level) by an attacker [11]. Therefore, we decided to avoid considering cache hits and to focus exclusively on cache miss events.

4.1.2 Observation Windows

We divide the entire observation period into time slots, which are handled as observation windows of HPC values that are inspected one by one. This allows discriminating among different execution phases. Such a discretization is applied to the number of elapsed clock cycles (which defines a constant unit among all running processes) rather than events such as retired instructions—they may warp the time slot depending on the executed instruction [12]. This window is preserved across context switches and is not shared among processes/threads, thus guaranteeing a coherent inspection of the execution flow.

The size of the time window is directly related to the overhead that the detection architecture introduces in the system because smaller slots imply more interrupts to be processed.

We also tune the size of the observation window at the system startup. To this end, we use an adaptative approach: if we observe a significant fluctuation in the data observed across two consecutive windows, we reduce the size of the window (up to a minimum threshold, which accounts for the overhead in the measurement). Conversely, if variations are minimal, we increase its size (up to a compile-time-based threshold). This approach allows, for long-lasting programs, to self-adapt to a suitable value, which can also characterize the usage of the memory hierarchy.

4.1.3 Suspecting Malicious Processes

After calculating the **Performance Monitoring Interrupt (PMI)** metrics, they are compared to the respective thresholds, thus determining if the driving-suspicion predicates hold. Based on the results of the inequalities, we deem a process as malicious or not. Obviously, the classification of a process cannot be made based on a single observation because we would have an excessive number of false positives considering that, during its execution, a process can assume different behaviours. For this reason, we have introduced a scoring system. The process's score will vary during execution as follows:

- the score is increased by α if the results of metrics/thresholds comparison show a behaviour similar to a side-channel attack;
- the score is decremented by β if the metrics do not detect any abnormal situation.

If the score reaches the value of a threshold γ, then the process becomes suspected. α, β, and γ are tunable hyperparameters of our model. These parameters are related to each other in the following way. α indicates how fast a process becomes suspected: the higher the value, the smaller is the number of positive evaluations of the metrics required to flag it as malicious. Conversely, β determines how fast a process that was (incorrectly) considered suspicious starts again to be deemed benign. α and β can be used to control our scoring system's responsiveness towards punctual activities (i.e., possibly malicious or not) exhibited within an observation window. In the general case, we assume $\alpha \geq \beta$ to allow for a prompt-enough detection of a malicious process. Conversely, γ directly controls when a process becomes flagged as malicious. To some extent, it indicates the amount of data that the system tolerates to leak before deeming a process as suspected.

4.1.4 Mitigation Strategies

Our kernel-based detection subsystem can flag a process as suspected. A suspected process is one for which we can implement mitigations. We note that this is not a destructive operation: even if we have incurred a classification error (i.e., a false positive), we enable mitigations that will not cause runtime errors (e.g., abnormal termination) in the wrongly suspected process. Indeed, we could only cause a performance slowdown. Nevertheless, considering the overall system, this slowdown will not be comparable to that observed if the mitigations we discuss here were activated by default for all processes. We have foreseen two families of mitigations: one pertaining to side-channel attacks in general and one related to transient execution vulnerabilities. The mitigations we put in place have

value independently of whether our own approach is used to detect the attacks, or the system would use other support to determine (potentially) malicious processes.

4.2 Results

Name	Same-Core	Cross-Core	Shared Memory	Measurement
EVICT + TIME [22] (taken from[17])	✓	✓	✗	time
PRIME + PROBE [23, 24] (taken from[18])	✓	✓	✗	time
PRIME + ABORT [25] (taken from[18])	✗	✓	✗	TSX
FLUSH + RELOAD [26] (taken from[18])	✓	✓	✓	time
FLUSH + FLUSH [27] (taken from[18])	✓	✓	✓	time
XLATE + TIME [28] (taken from[18])	✓	✓	✓	time
XLATE + PROBE [28] (taken from[18])	✓	✓	✓	time
XLATE + ABORT [28] (taken from[18])	✓	✓	✓	TSX
Meltdown (taken from [19])				
Spectre (taken from [20])				
Foreshadow [29, 30] (taken from [21])				

Figure 8: Overview of considered cache side-channel attacks and references to the used implementations

We have carried out an experimental assessment relying on multiple generations of Intel CPUs, namely using the following processors:

- i7-6700HQ 4x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 6MB 12-way;
- i7-7600U 2x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 4MB 16-way (with TSX);
- i5-8250U 4x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 6MB 12-way;
- i7-9750H 6x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 16MB 16-way;
- i7-10750H 6x (SMT) L1 64KB (I,D) 8-way, L2 256KB, shared L3 12MB 16-way.

To set up the thresholds and observation windows used by our detection system, we have run versions of the attacks listed in Figure 8, as well as the following set of benignware applications: (1) Firefox, with both textual pages, multimedia content access, and browser benchmarks such as JetStream2; (2) VLC, with both large and short videos and random skip of video portions, as well as repositioning; (3) Evince Reader, with both small and large size pdf files, and random skip of pages; (4) gedit for editing textual files of different sizes and random positioning onto the file portion to be edited; (5) all the kernel-level threads operating within the Linux kernel.

4.2.1 Accuracy of HPC Events

We evaluated HPCs accuracy in terms of both over-counting and determinism by comparing the data collected from HPCs with data obtained from software instrumentation—results are reported in Figure 9. We relied on a basic (single thread) benchmark for this experiment, which computes the

first x prime numbers, where x is a user-defined parameter. We used `cachegrind` [13] as a baseline, which automatically detects the underlying cache structure and builds an equivalent cache model while executing the program. With `cachegrind`, we can compare the results related to memory accesses and cache misses. Nevertheless, L3 cache misses, L2 filled lines (it counts opportunistic events at cache line grain and includes prefetcher activity), and TLB miss (we use a specific event that requires the emulation of a second-level TLB) are not available. For these events, we compared the HPCs values of several runs to compute the determinism degree of this source. The results in Figure 9 experimentally confirm that, although HPCs could be subject to reliability errors, we have selected more stable and portable events across different architectures. Although the L1 miss Err value may be a wake-up call to the reader, it is consistent among the tested architectures and the HPCs variation coefficient. This result stems from `cachegrind`'s inability to model all the hardware counterpart's internal details vendors do not disclose.

	i7-6700HQ				i7-7600U				i5-8250U			
	HPCs	SW	Err	HPCvar	HPCs	SW	Err	HPCvar	HPCs	SW	Err	HPCvar
loads	4494K	4744K	5.2%	~0%	4494K	4744K	5.2%	~0%	4494K	4744K	5.2%	~0%
L1 miss	400K	308K	29%	2.0%	515K	308K	66%	2.9%	521K	308K	69%	1.6%
L3 miss	8557	-	-	6.4%	7798	-	-	3.9%	6539	-	-	3.6%
L2 lines	185K	-	-	7.9%	221K	-	-	3.1%	224K	-	-	4.7%
TLB miss	9168	-	-	~0%	9382	-	-	5.7%	8981	-	-	1.9%

	i7-9750H				i7-10750H			
	HPCs	SW	Err	HPCvar	HPCs	SW	Err	HPCvar
loads	4494K	4744K	5.2%	~0%	4495K	4744K	5.5%	~0%
L1 miss	513K	312K	64%	2.2%	517K	308K	67%	1.8%
L3 miss	8064	-	-	4.0%	3725	-	-	7.7%
L2 lines	227K	-	-	2.9%	226K	-	-	5.5%
TLB miss	9321	-	-	2.4%	9301	-	-	3.0%

Figure 9: Comparison between HPCs and Software Instrumentation on all the architectures. Err represents the distance (%) between HPCs and SW while HPCerr shows the HPCs variation coefficient

4.2.2 Accuracy of System-Wide detection approach

To assess the capabilities of our detection system, we have performed a system-wide experimental evaluation by building sets of benignware and malware applications. The former relies on the Phoronix Test Suite [14], from which we selected 156 benchmarks (configured with different inputs) showing various behaviors and load profiles. Conversely, to build the set of malicious applications to exercise our solution's capability to detect side-channel attacks, we have not found access to real-world malware of this kind. As a consequence, we have crafted such malicious applications starting from the stress-ng suite [15]. We injected side-channel attacks (based on the implementations reported in Figure 8) into various benchmarks of the suite, generating a set of 100 malicious applications. The side-channel routine is placed within the benchmark stress function. The attack is anyhow enabled only after a random delay and, after its activation, the side-channel procedure executes with a specific probability—we set this probability to 10%. By introducing these sources of uncertainty, we increased the non-determinism degree that attacks may exploit in realistic scenarios. As described, the behavior of our detection system highly depends on the α, β, and γ hyperparameters. We set α and β to 1 for the entire experimental phase while varying γ to evaluate the detection according to different threshold levels. As discussed, α and β represent the rates that regulate the score progression of each process in the system. By setting $\alpha = \beta = 1$, we are identifying a critical scenario for our detection system, as we slow down the detection of malicious applications while reducing the possibility for a benign application to "recover" from spurious actions being detected as malicious. At the same time, by varying γ, we somewhat change the responsiveness to an undergoing attack.

Figure 10: Detection accuracy evaluation for different values of γ (α, β = 1). P4 and S1 mark attacks that are detected by these predicates while OK indicates normal processes

Figure 10 shows the results of the detection accuracy as confusion matrices. The standard benchmarks (i.e., with no side-channel attack injected) are labelled as OK, while S1 and P4 indicate the attacks detected by the corresponding predicate. Confusion matrices with γ = 1 illustrate the behavior of our detection system as if the scoring system were not available. In this configuration, any application becomes suspected after a single violation of any metric. As we can observe, the

number of false positives is non-minimal, and on the i7-6700HQ it is even higher than real negatives. Overall, the benchmarks which have been wrongly suspected are the ones which either: i) involve a large number of forks and therefore propagate the information associated with the measures across a large number of processes; ii) implement data processing or machine learning algorithms iii) are memory-intensive scientific applications or explicitly test the memory hierarchy. Nonetheless, the number of false positives quickly decreases as the value of γ increases. Indeed, this is related to the fact that subsequent observations can filter out any potential spike in applications' activity without prematurely marking the process as suspected. This trend matches exactly our expectations, also validating the viability of the scoring system. Our experiments did not report any false-negatives detection. The approach we have proposed well fits scenarios in which a higher level of security is desired. However, the system is still prone to performance optimization under very low-security risks. Moreover, by design, the tuning mechanism aims to reduce the likelihood of experiencing false negatives at the cost of slightly increasing the number of false positives. Nevertheless, if γ is set to a suitably high value, this number becomes negligible. By definition, a detection system is not a predictor, but it reacts to some events and makes decisions according to its model. Indeed, such a characteristic is crucial. Before classifying a malicious process as suspected, we expect part of its attack to have been executed—at least, the portion required to generate an identifiable pattern by our detection system. Typical side-channel attacks rely on a preliminary preparation phase (e.g., probing the cache) during which no data is actually read. If our detection system can detect a side-channel attack during this preparation phase, the attacker will not be able to read any data. Conversely, if the detection system identifies the attack during its extraction phase, then the attacker might read some amount of information. Overall, the amount of data that an attacker can read even if our detection system is active is an important metric to assess the accuracy of our system-wide detection approach. Therefore, we have carried out an experiment to quantify the amount of data that a malicious process can read before its detection. In this experiment, the attacker shares a chunk of read-only memory with the victim and tries to leak information by mounting a side-channel attack on a byte-by-byte basis. Concurrently, the victim repeatedly reads the shared buffer with some delay among subsequent accesses, generating all the conditions to perpetrate the cache-based attack. In this experiment, we have set the secret's size to be extracted to 256 bytes—this is a non-minimal buffer corresponding to the size of a large **Advanced Encryption Standard (AES)** key. We have run this attack with our detection system turned off, which forms the baseline for our assessment. Figure 11 shows the results of this experiment with detection capabilities turned on with various values of γ and different victim's read rates. With γ = 100, our approach can detect the attack before it extracts a significant fraction of the data extracted when no detection was active, but only when the victim reads with minor delays. By decreasing γ, the percentage of correctly extracted data is reduced. Although the reader may think that by increasing the victim's read rate the detection may fail to identify the attack promptly, our results show that the percentage of extracted bytes decreases for very high frequency reads on all examined architectures. This phenomenon is due to side-channel attacks being more sensitive to the noise generated when the activity in the system increases.

4.2.3 Performance Assessment

We have studied the performance improvement that we can obtain with our monitoring proposal. To quantify the performance benefit of our approach, we have again relied on the Phoronix Test Suite, selecting a set of benchmarks that interacts with the system in different ways, according to the following classes of behavior:

A. intensive disk I/O operations (compilebench);
B. pressure on the scheduler and context switch operation, also considering multithreaded applications (hackbench, ctx_clock);
C. a large number of system call invocations, such as fork, exec, and those related to memory management (OSBench);
D. high usage of the network socket API (sockperf);
E. high usage of the GNU C Library APIs (glibc-bench)

F. Complex workloads, related to browsers and databases (selenium, sqlite-speedtest, Apache).

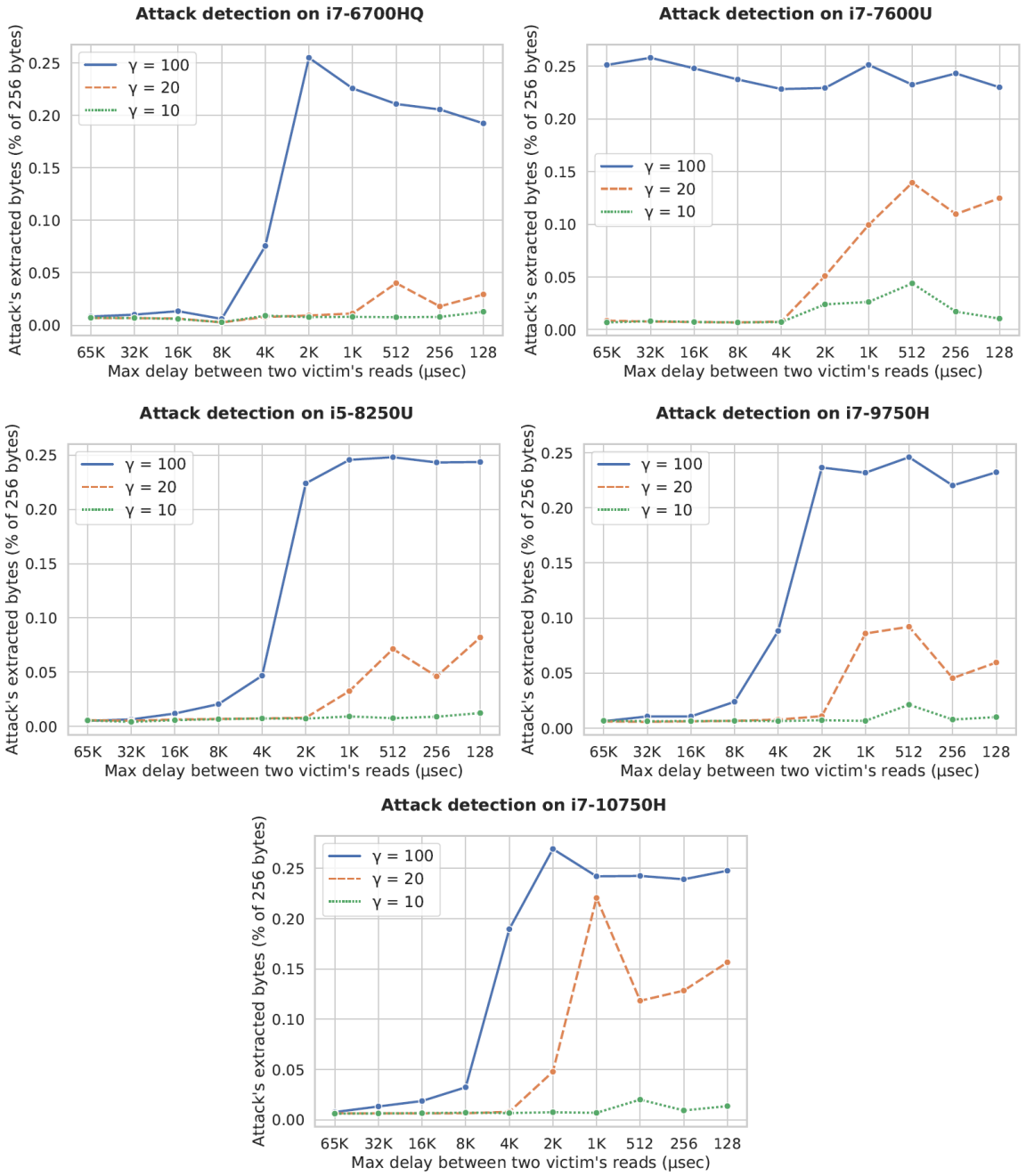

Figure 11: Percentage of a 256-byte secret that an attack can correctly extract before being detected, for different values of γ (α, β = 1) and victim's read rates.

We also note that selecting these benchmarks allows profiling multiple classes of applications, namely CPU- bound ones (in userspace) or applications that repeatedly interact with the kernel, forcing the application to make a substantial number of mode switches. Given the implementation of software patches, we should influence the performance of the considered applications. No side-channel attack has been mounted in this experiment.

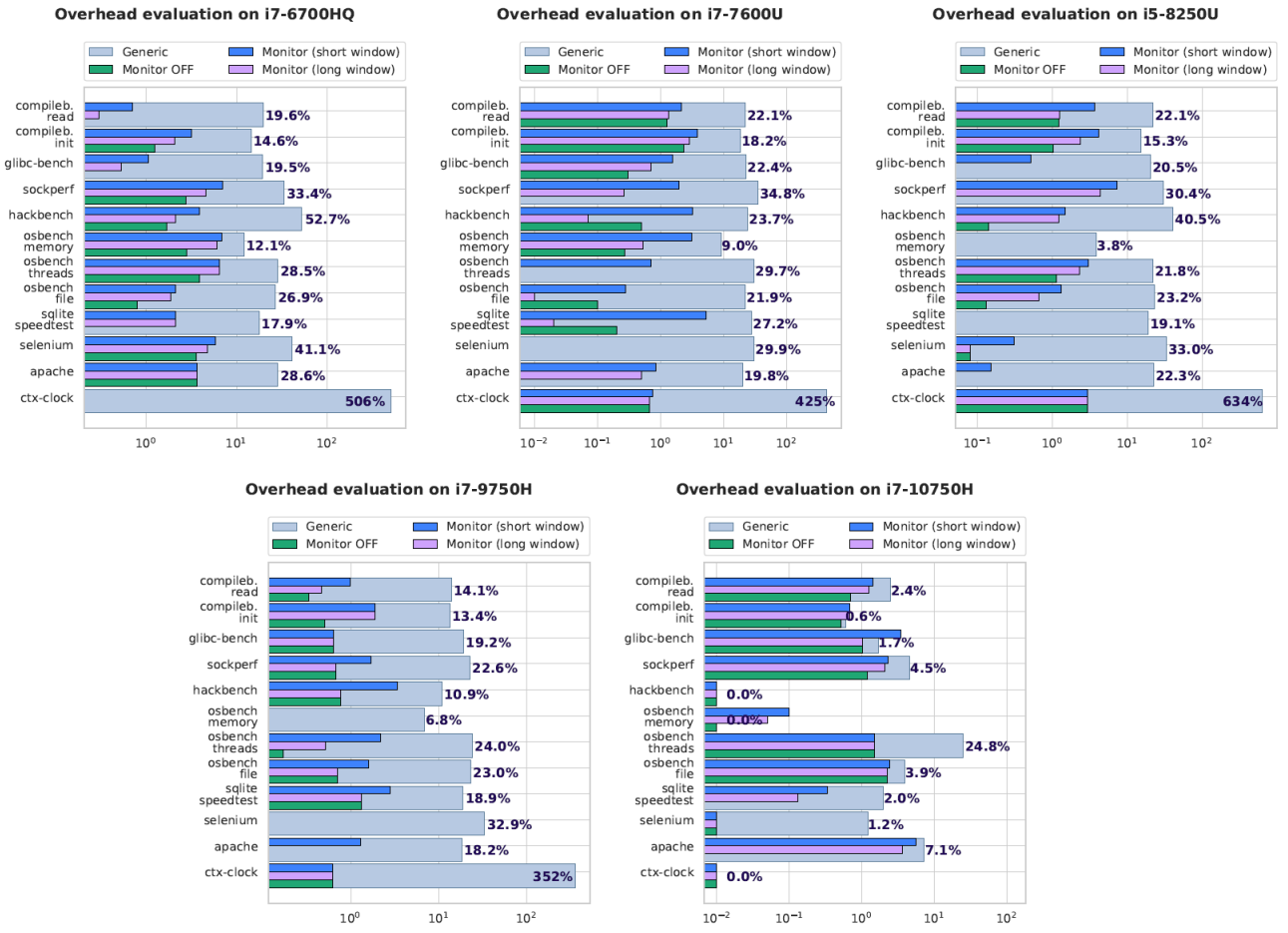

Figure 12: Performance Effects of the HPC-based Monitoring System on different Architectures (log scale on the x-axis).

These benchmarks were run in the four following scenarios to evaluate the performance impact of the system-wide detection scheme, also accounting for the effect of the observation window:

A. Mainline kernel 5.4.145 with KPTI, retpolines, SSB mitigations, and all the patches discussed in Section 6 enabled by default for all processes—referred to as Generic in the plots.
B. Kernel 5.4.145, with our support for dynamic patching, but with system-wide monitoring disabled—referred to as Monitor OFF in the plots.
C. Kernel 5.4.145, with our system-wide detection scheme activated, with an observation window set to 2 20 clock cycles, which was the minimum observation window value considered by the adaptative approach in our setup—referred to as Monitor (short window) in the plots.
D. Kernel 5.4.145, with our system-wide detection scheme activated, with an observation window set to 2 24 clock cycles, which was the maximum observation window value considered by the adaptative approach in our setup—referred to as Monitor (long window) in the plots.

The results for the benchmarks in these configurations are reported in Figure 12, where we show the overhead compared to the mainline kernel 5.4.145 with no active patch, which is vulnerable to all the discussed attacks—values are averaged over three different runs. By the results, we can observe that the `Monitor OFF` approach offers a performance slowdown with respect to the Generic configuration, which is up to 4 orders of magnitude lower while showing an overhead over the unpatched mainline kernel lower than 4% on all architectures and for all application classes. This

means that the supports which we have introduced in the kernel to enable/disable at runtime the various security patches are lightweight and non-intrusive. Conversely, the overhead of the Monitor configuration over the `Monitor OFF` configuration is negligible. It is interesting to note that the impact of the window length is minimal: considering that they are related to the maximum/minimum values supported by our system, this experiment shows that the expected overhead, also accounting for the adaptative optimization of the window, is reduced. Of course, this reduced overhead is coupled with the increased security level which our proposal can offer. Overall, this is additional evidence of the viability of our proposal. A similar trend can be observed for all tested architectures (except for the i7-10750H) and all classes of applications, although with different relative ratios. This is an indication of the stability of our approach with respect to the performance of applications. The results on the i7-10750H processor do not match the other models' behaviour. This is because Intel, starting from the 10th generation of its processors, introduced design changes to patch some hardware vulnerabilities. Consequently, the Linux kernel does not require enabling all the software patches (such as KPTI) on these processors to mitigate the performance slowdown. Nevertheless, our approach can still detect side-channel attacks on more modern architecture for which a hardware patch has not been proposed, with reduced overhead.

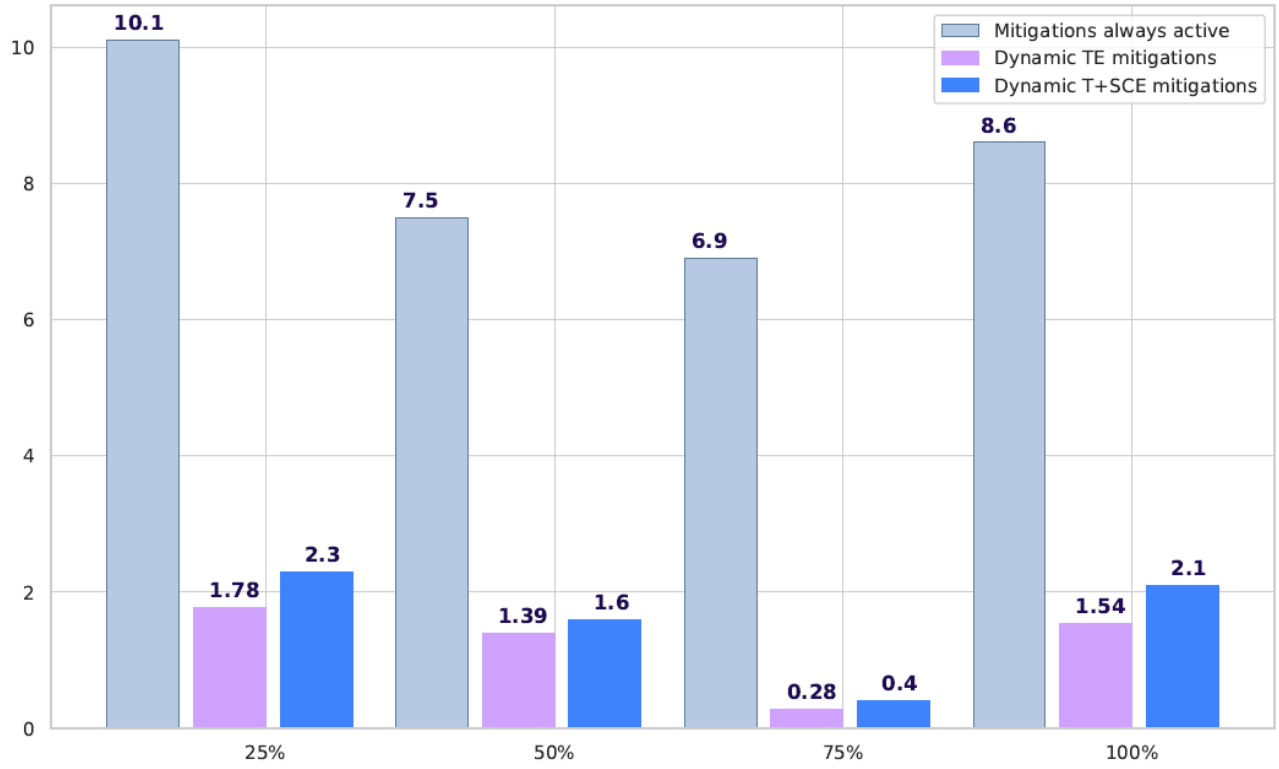

Figure 13: Performance Penalties by Mitigations on the i5-8250U.

The last experiment we present—the data reported in Figure 13—relates to an assessment of the overhead due to transient execution mitigations and side-channel mitigations, also when there is significant interference with *benignware* on the same CPU cores. For this experiment, we only report data taken on the i5-8250U machine for the sake of space. In any case, the results on the other architectures show trends that are perfectly comparable with the data reported on this CPU. We have launched a number of benchmarks taken from the Phoronix Test Suite equal to the number of available cores on the considered processor.

Each benchmark has been statically pinned to one CPU core. We then varied the number of malicious applications, pinned them to specific CPU cores, and ran them concurrently with the *benignware* benchmarks. This setup stress-tests also the per-process detection/mitigation capabilities of our system. We report data associated with the system run with all transient execution mitigations always active (Mitigations always active in the plot), with transient execution mitigations

activated only for suspected processes (Dynamic TE mitigations in the plot), and with transient execution/side-channel mitigation countermeasures started only for suspected processed (Dynamic TE+SC mitigations in the plot). The applications have been selected to avoid any false positive/negative. As in the previous experiment, we report the overhead as the percentage increase over an execution in which no mitigation at all (neither static nor dynamic) is present in the system. By the result, we observe again that enforcing dynamic mitigations provides a significant overhead reduction, as high as 95%. As expected, the overhead incurred when also SC mitigations are active is higher. Of course, depending on the system's configuration, the user can determine what set of mitigations should be enforced upon the detection of a malware application.

4.3 Conclusions and Future work

The metrics we have devised have allowed us to detect attacks with a negligible percentage of false positives and no false negatives. The data have been collected on different flavours of x86 Intel CPUs and with a comprehensive set of benchmark applications (either *benignware* or malware). Based on the comprehensive architecture we have presented in this paper, we plan to expand the set of detection metrics as future work to account for other kinds of memory-based attacks, such as Rowhammer [6]. Furthermore, we plan to port our solution to processors from vendors other than Intel, e.g. AMD. Concerning the effects of our mitigation strategies, we additionally plan to conduct an experimental assessment to show the impact on our approach's power consumption.

Chapter 5 Steady/ProjectKB – SAP

This chapter provides additional information regarding functionalities that have been developed and tested in the context of SPARTA, both for Project KB and Eclipse Steady.

5.1 Vulnerability creation and end-to-end import

This test covers the creation of vulnerability information in Project KB as well as its import into Eclipse Steady and, from there, into VulnEx.

Procedure:

The following test steps have been performed:

1) Manual creation and digital signature of a vulnerability statement for CVE-2021-36373 and CVE-2021-36374 in a dedicated branch “sparta-test” of a fork of Projekt KB (covering Project KB’s software requirements SR1, SR2)
2) Manual change of a vulnerability statement to track changes (Project KB SR-4)
3) Configuration update of Steady component kb-importer to read statements from that fork and branch (Project KB SR3, Steady SR3)
4) Restart of Steady’s Docker Compose application to make configuration change effective
5) Observation of logs and Steady’s Web frontend to check whether the new vulnerability has been automatically imported

Result: Success

Figure 14 and Figure 15 show the creation and change of the vulnerability statement for CVE-2021-3673. The “verified” icon indicates that the commit has been digitally signed using the committers private key.

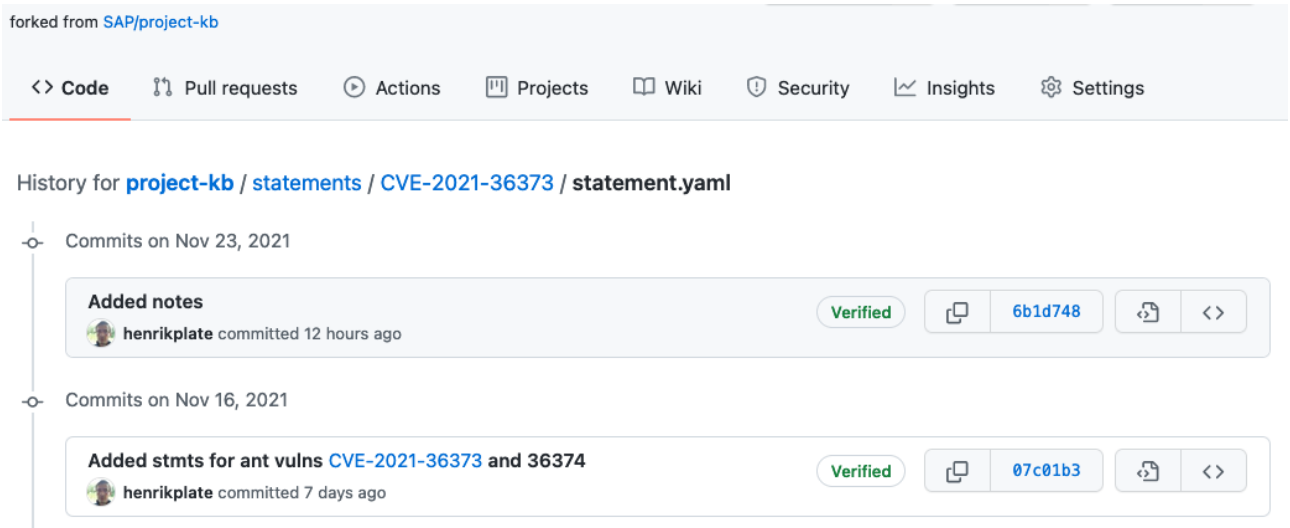

Figure 14: Creation and change of vulnerability statement in fork of Project KB

Figure 15: Change as part of commit 6b1d748

Figure 16 shows a log file snippet demonstrating that the configuration for Steady component kb-importer has been taken into account.

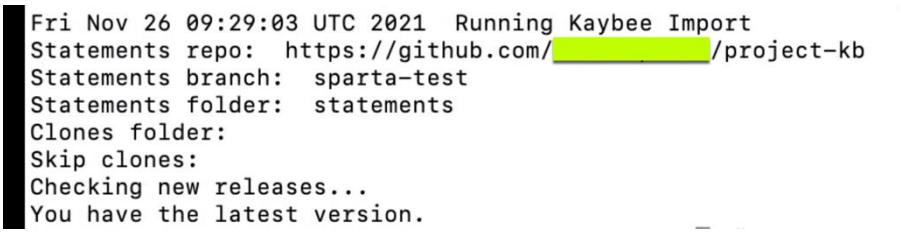

Figure 16: Screenshot showing the use of the new configuration setting

Finally, Figure 17 shows Steady's Web frontend before the import of the above mentioned vulnerabilities from the dedicated branch in a clone of Project KB, whileFigure 18 , Figure 19 and Figure 20 show the Web frontends of Steady and VulnEx after the completion of the end-to-end import.

Note that the application did need to be scanned another time, the information collected during past scans sufficed Steady to discover that it is affected also by the new vulnerabilities.

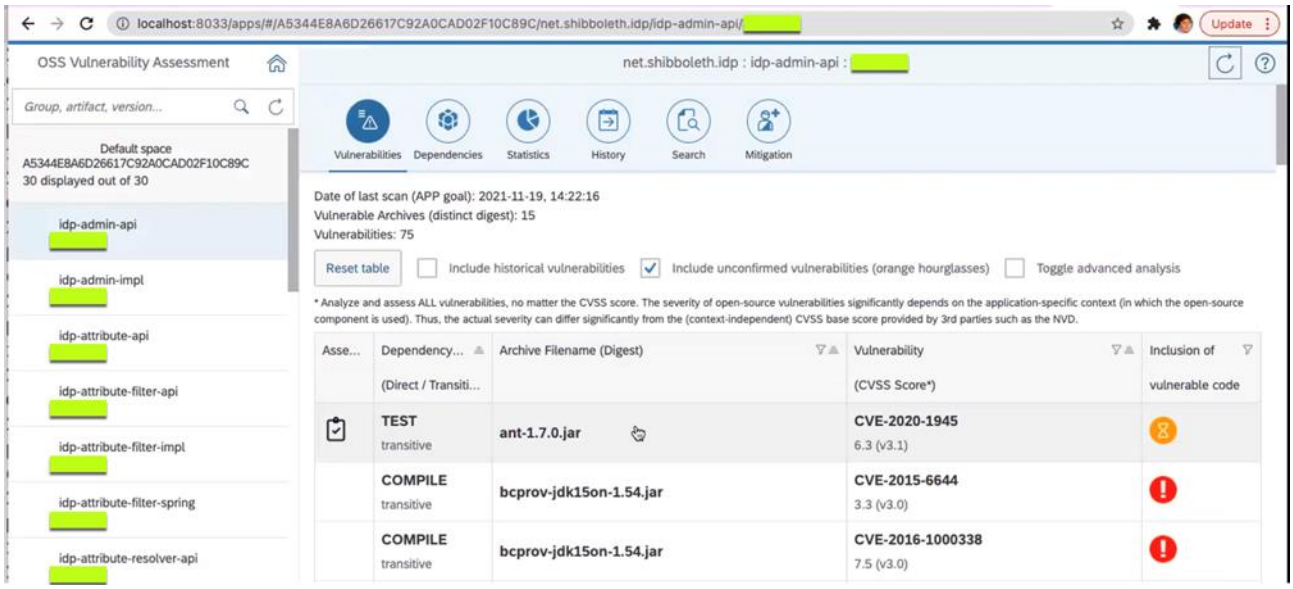

Figure 17: Screenshot of Steady before import of new CVEs

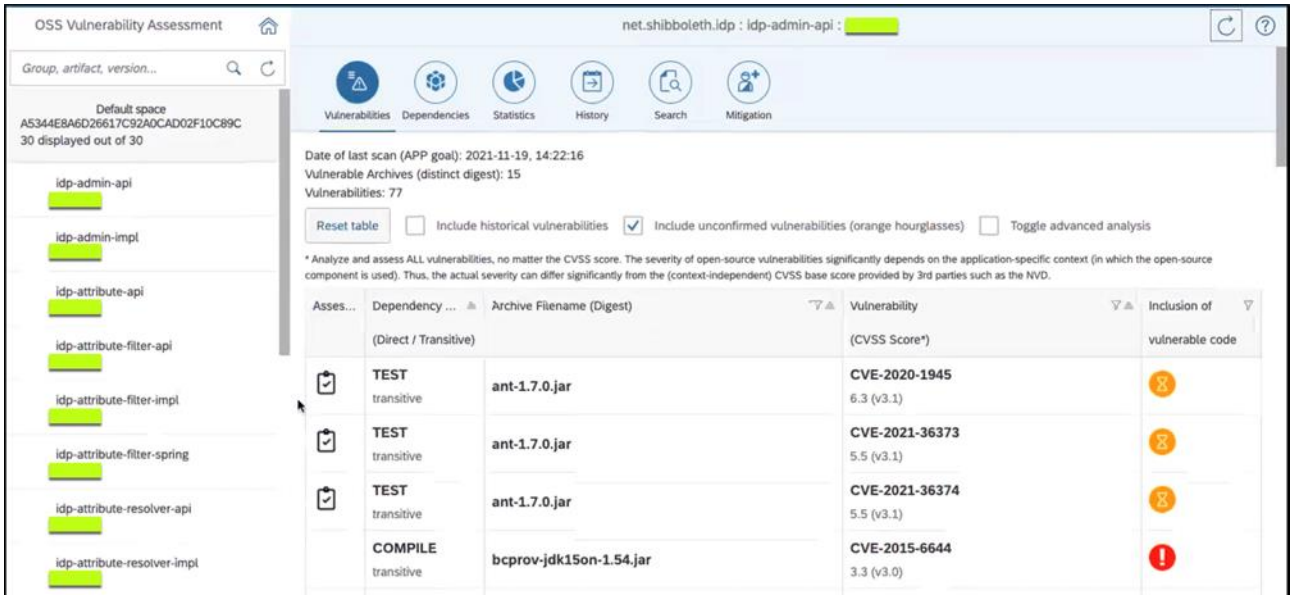

Figure 18: Screenshot of Steady after import of new CVEs from Project KB fork

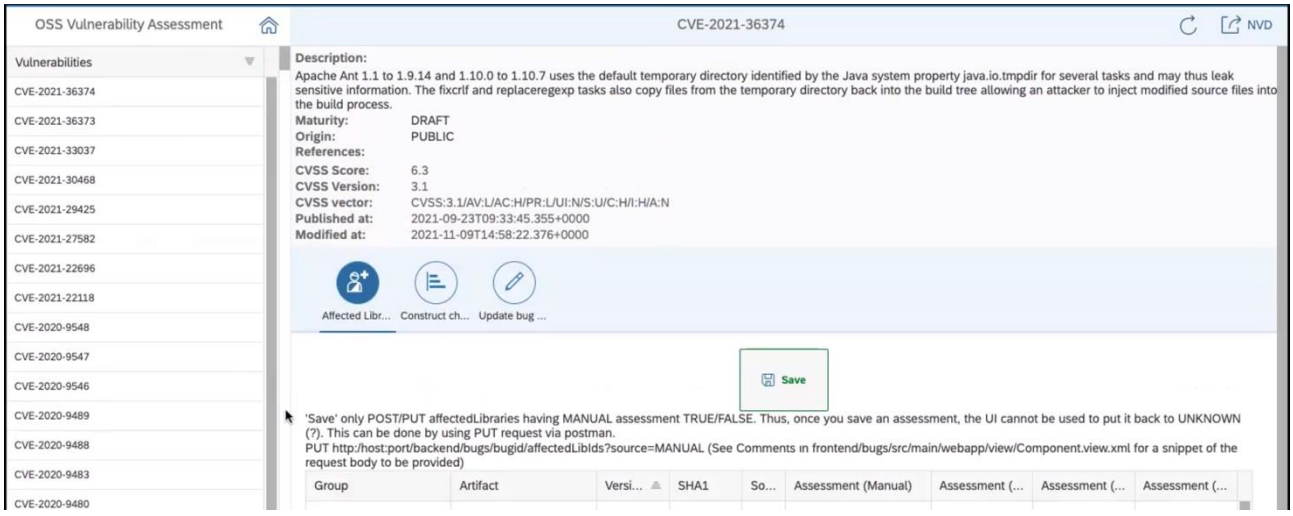

Figure 19: Screenshot of Steady's bug frontend with imported CVEs

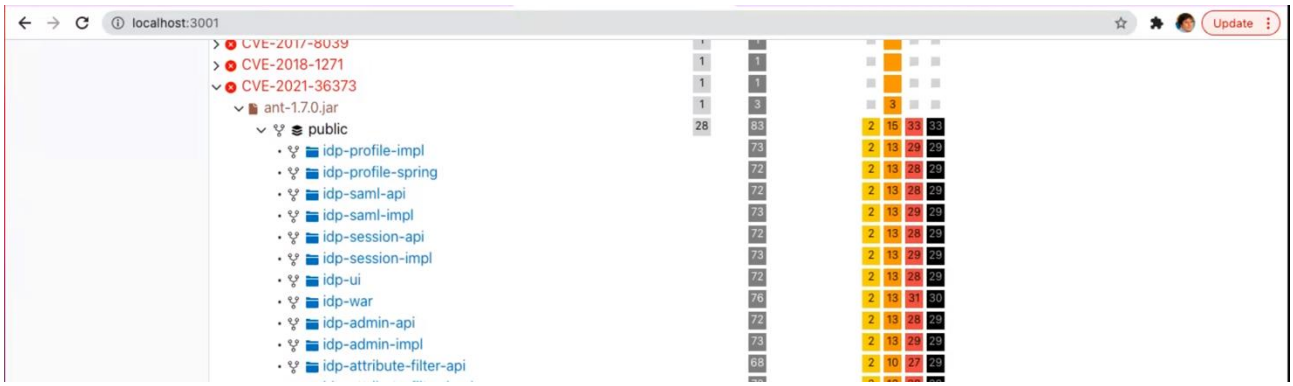

Figure 20: Display after import of new CVEs from Steady to VulnEx

5.2 Comparison of Java source code and bytecode (Steady SR1)

This test covers the implementation of the "checkcode" goal (Steady's software requirement SR1), which has the goal to further improve the detection rate of Steady, for cases where Steady was not yet able to determine whether a given method's body correspond to the vulnerable or the fixed version. See D5.3 for a description of the approach.

Procedure:

The following test steps have been performed:

1) Identify an archive with method bodies identical to a non-resolved candidate vulnerability (orange hourglass) in the application under test
2) Import the library into the Steady backend
3) Assess the imported library as vulnerable using Steady's bug frontend
4) Run the "checkcode" goal on the IDP application
5) Open the Steady Web frontend for the scanned application and verify that the orange hourglass disappeared (and switched to a red exclamation mark, herewith signaling a vulnerability)

Result: Success

1) We identified that the library with GAV (org.apache.servicemix.bundles:org.apache.servicemix.bundles.ant:1.7.0_1) contains the same vulnerable code related to CVE-2020-1945 as the dependency (org.apache.ant:ant:1.7.0), which is one of the IDP's dependencies showing an orange hourglass (cf. Figure 21).
2) The library was created in the Steady backend by uploading JSON via a CURL POST request.
3) We opened Steady's bug frontend, and the new servicemix archive was shown, and its status was manually changed to "vulnerable" (cf. Figure 22).
4) When running the checkcode goal via command line on the IDP application, Steady discovered that the unassessed code in (org.apache.ant:ant:1.7.0) resembles the one in the assessed servicemix library, and applied the same "vulnerable" assessment to it (cf. Figure 23).

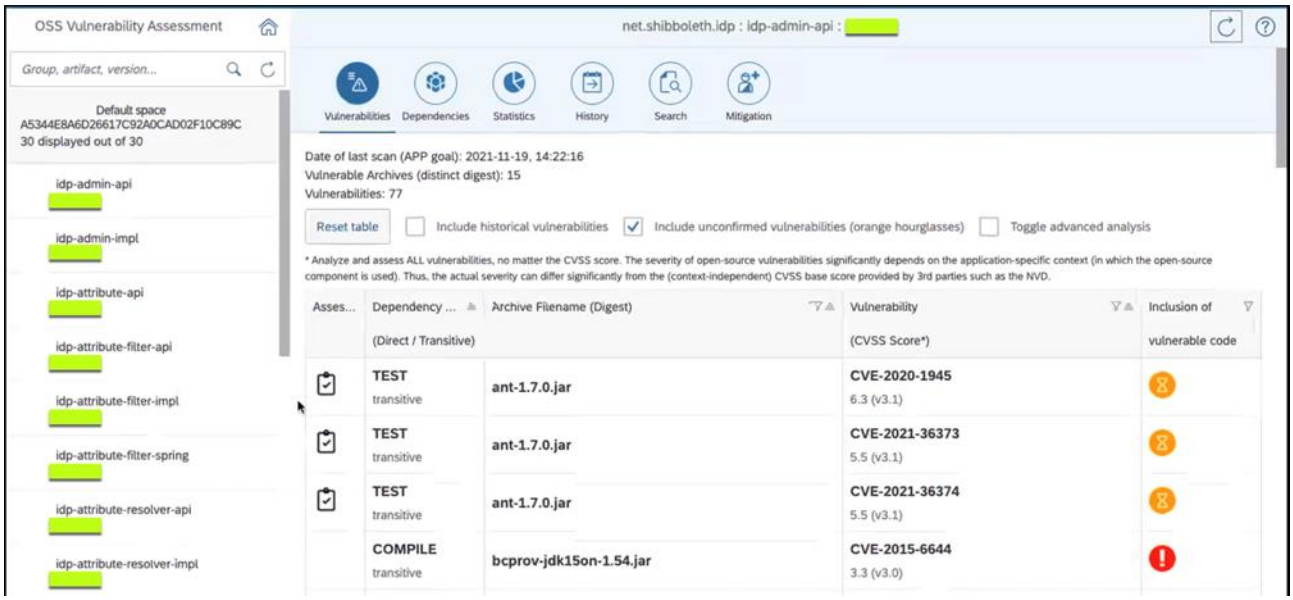

Figure 21: Archive ant-1.7.0.jar potentially vulnerable to CVE-2020-1945

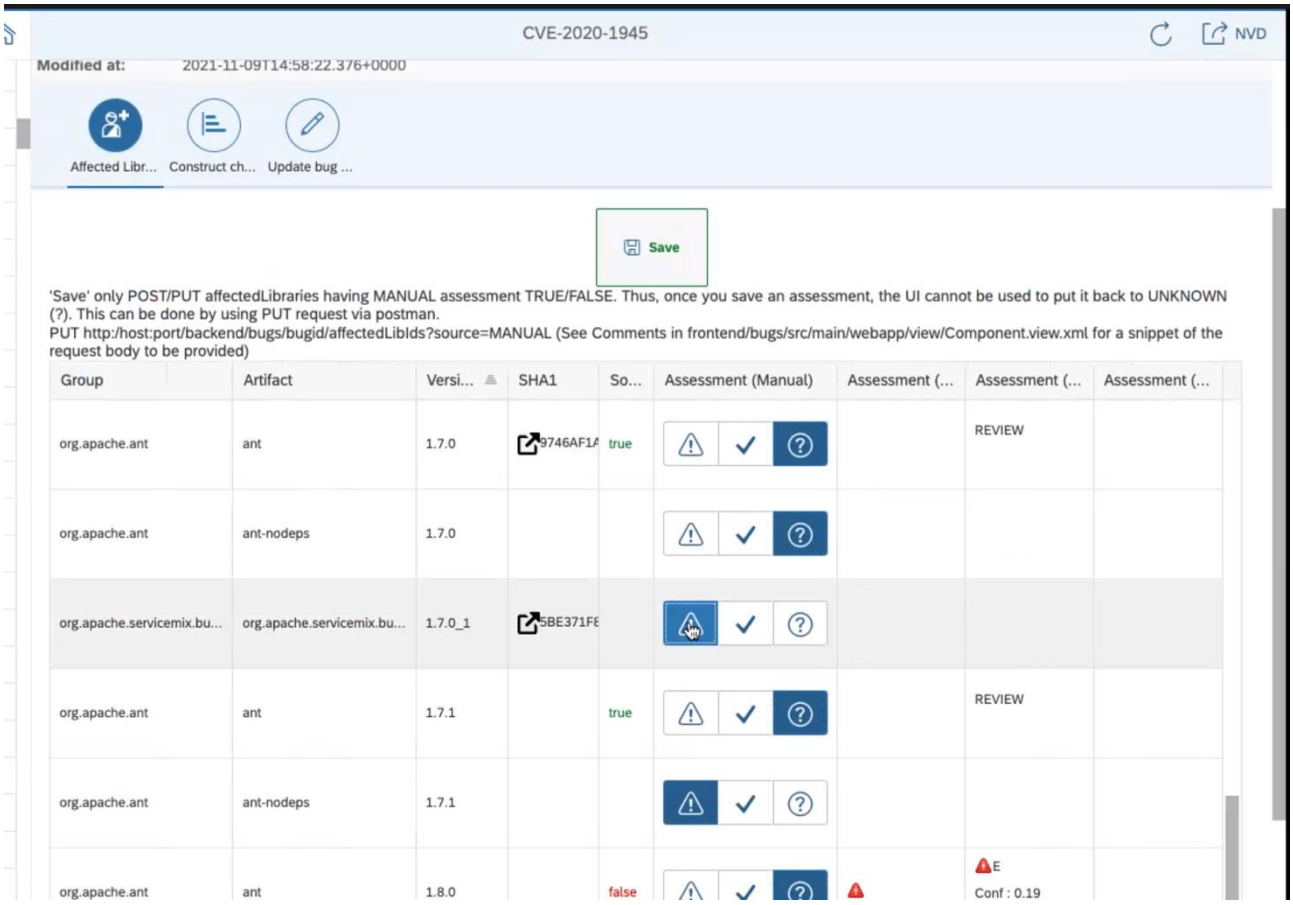

Figure 22: Marking servicemix re-bundle as vulnerable (in Steady's bug frontend)

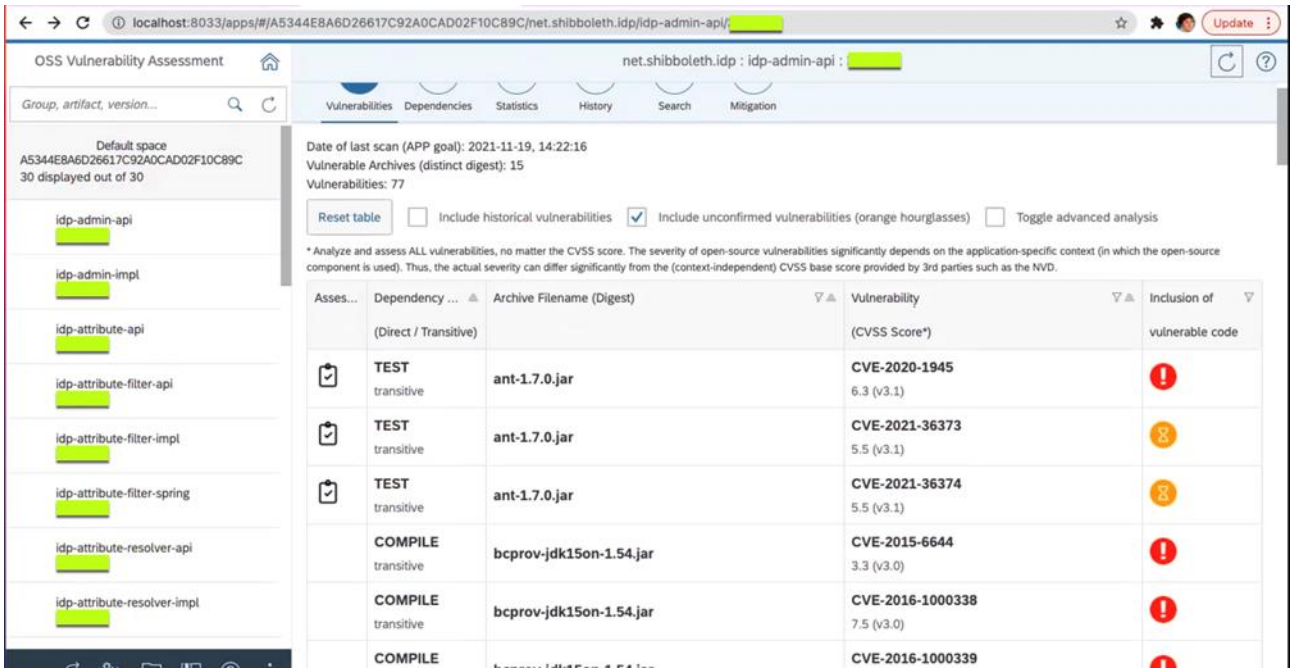

Figure 23: Automated change of CVE-2020-1945 through running of checkcode goal

5.3 Light-weight scan client (Steady SR2)

This test covers the implementation of a light-weight version of Eclipse Steady's Docker Compose environment (Steady's software requirement SR2), the main objectives being to facilitate the use of Steady without needing to operate a heavy backend, and to support quick experiments with an easy setup.

The implementation consisted of rewriting the Docker Compose file such that the services are grouped into core services, UI services and service related to the vulnerability database (VDB). Both UI and VDB services can be started and stopped as needed, which reduces the footprint of the entire Docker Compose application. Moreover, one service was entirely removed and re-implemented. The management of those service groups is done via a new install and start scripts, which abstracts Docker specifics from the user.

Procedure:

The following test steps have been performed:

1) Install Steady in a fresh environment
2) Start all services
3) Monitor resource consumption
4) Stop all but core services
5) Monitor and compare resource consumption

Result: Success

The Figures 24-28 show screenshots of the above-described test procedure. In particular, Figure 26 and Figure 28 demonstrate the reduced footprint of running just 3 core services compared to all 9 services.

```
[redacted] % ./setup-steady.sh -t f99ce2c -d docker
Installing Eclipse Steady...
    from https://raw.githubusercontent.com/eclipse/steady/f99ce2c/docker
    into /Users/[redacted]/Documents/ossa/steady/docker/docker/
Created default client configuration /Users/[redacted]/.steady.propertiesInstallation completed, Steady can be started using ./docker/start-steady.sh

IMPORTANT: Before starting it for the first time, change the default passwords in ./docker/.env

Press <a> to start all of Steady's Docker Compose services (or any other key to skip execution):
Execution skipped. Check startup options with ./docker/start-steady.sh --help)
```

Figure 24: Steady installation using the new install script

```
[redacted] % ./start-steady.sh -s all
[+] Running 9/9
 ⠿ Container steady-rest-lib-utils      Started
 ⠿ Container steady-cache               Started
 ⠿ Container steady-postgresql          Started
 ⠿ Container steady-rest-backend        Started
 ⠿ Container steady-frontend-apps       Started
 ⠿ Container steady-patch-lib-analyzer  Started
 ⠿ Container steady-kb-importer         Started
 ⠿ Container steady-frontend-bugs       Started
 ⠿ Container steady-haproxy             Started
Started all of Steady's Docker Compose services

Scan your Maven project as follows:

    mvn org.eclipse.steady:plugin-maven:3.2.0:app

Point your browser to:

    http://localhost:8033/apps to see the results of your application scans
    http://localhost:8033/bugs to see all vulnerabilities imported from Project KB into Steady's database

Find more information at https://eclipse.github.io/steady
```

Figure 25: Start-up of all 9 Steady services

```
CONTAINER ID   NAME                       CPU %     MEM USAGE / LIMIT     MEM %     NET I/O           BLOCK I/O         PIDS
8ddb10b905ea   steady-haproxy             0.20%     4.145MiB / 1.938GiB   0.21%     426kB / 62.6kB    2.61MB / 0B       9
8750d511fa3f   steady-kb-importer         18.49%    85.73MiB / 1.938GiB   4.32%     78.4MB / 1.72MB   58.2MB / 0B       12
c52bc6aa1a76   steady-patch-lib-analyzer  0.00%     740KiB / 1.938GiB     0.04%     1.13kB / 0B       8.19kB / 0B       2
a3ddeab4c6e7   steady-frontend-bugs       0.11%     108.5MiB / 1.938GiB   5.47%     14.1kB / 153kB    11.8MB / 32.8kB   32
25c67d4a2185   steady-frontend-apps       0.29%     107.8MiB / 1.938GiB   5.43%     14.1kB / 178kB    20MB / 32.8kB     32
42398171076a   steady-rest-backend        1.26%     361MiB / 1.938GiB     18.19%    177kB / 144kB     51.5MB / 0B       50
f82288d8e751   steady-cache               0.05%     42.84MiB / 1.938GiB   2.16%     33.9kB / 36.6kB   4.87MB / 8.19kB   266
b1bd9a9c4cc0   steady-rest-lib-utils      0.14%     293MiB / 1.938GiB     14.76%    13.3kB / 17.9kB   41.1MB / 0B       42
f2ee5fe5357e   steady-postgresql          0.03%     78.23MiB / 1.938GiB   3.94%     73.1kB / 151kB    11.6MB / 61.4kB   57
```

Figure 26: Docker stats for all 9 Steady services

```
[redacted] % ./start-steady.sh -s core
[+] Running 3/3
 ⠿ Container steady-frontend-bugs  Stopped                    0.7s
 ⠿ Container steady-frontend-apps  Stopped                    0.7s
 ⠿ Container steady-cache          Stopped                    0.3s
[+] Running 3/3
 ⠿ Container steady-patch-lib-analyzer  Stopped              10.2s
 ⠿ Container steady-kb-importer         Stopped              10.3s
 ⠿ Container steady-rest-lib-utils      Stopped              10.2s
[+] Running 3/3
 ⠿ Container steady-postgresql    Running                     0.0s
 ⠿ Container steady-rest-backend  Running                     0.0s
 ⠿ Container steady-haproxy       Running                     0.0s
Started Steady's core Docker Compose services

Scan your Maven project as follows:

    mvn org.eclipse.steady:plugin-maven:3.2.0:app

Find more information at https://eclipse.github.io/steady
```

Figure 27: Start-up of only 3 core services (required for application scans)

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
8ddb10b905ea	steady-haproxy	0.34%	4.438MiB / 1.938GiB	0.22%	997kB / 257kB	3.03MB / 0B	9
42398171076a	steady-rest-backend	0.22%	376.6MiB / 1.938GiB	18.98%	930kB / 862kB	54.7MB / 0B	58
f2ee5fe5357e	steady-postgresql	0.01%	83.67MiB / 1.938GiB	4.22%	310kB / 644kB	16.6MB / 197kB	57

Figure 28: Docker stats for 3 core services

Chapter 6 Legitimate Traffic Generation system (LTGen) – IMT

This section deals with the redesign of the LTGen generator using a new approach in relation with traffic distribution learnt from real traffic datasets. Going beyond synthetic dataset generation, the new approach proposes to reconstruct realistic network packets from a limited number of features, at first. This prototype needs to be generalized to a greater number of features. This will enable evaluators to produce synthetic traffic to replay against intrusion detection systems, without incurring privacy issues, and having to re-dimension traffic captures.

6.1 Introduction

A flow is a communication between two machines on a network. In the Intrusion Detection System literature, several datasets propose different features to characterize a flow. This flow characterization from a set of features can be called a flow profile. The aim of this project is to generate packets to form a flow that will respect a given flow profile.

The first part will introduce the features selection process that we performed. Then the second part will present the tools used to generate some packet flows. And the last part will report on and discuss the generation experiments.

6.2 Flow features set

We first analyze a sample of datasets from the Intrusion Detection System literature in order to gather a wide range of different flow features. The features of four popular datasets were extracted: 42 flow features from BOT_IoT [31] ; 49 features from UNSW-NB15 [32]; 44 flow features from TON_IoT [33] ; and 80 network traffic features from CICIDS2017 [34].

From these datasets, we extracted common and similar flow features to obtain a limited list of features to model flows to be generated (see Table 2).

Flow features set
Source IP address
Source port number
Destination IP address
Destination port number
Flow duration
Protocol
Source to destination bytes
Destination to source bytes
Source to destination packet count
Destination to source packet count

Flow features set
Source bits per second
Destination bits per second
Down/Up Ratio
Number of flow packets per second
Minimum packet length from Source
Maximum packet length from Source
Mean packet length from Source
Standard deviation packet length from Source
Minimum packet length from Destination
Maximum packet length from Destination
Mean packet length from Destination
Standard deviation packet length from Destination
Flow IAT Min*
Flow IAT Max
Flow IAT Mean
Flow IAT Std
Forward IAT Min
Forward IAT Max
Forward IAT Mean
Forward IAT Std
Forward IAT Total
Backward IAT Min
Backward IAT Max
Backward IAT Mean
Backward IAT Std
Backward IAT Total

Table 2: Set of the selected flow features

We then reduced this list from 37 features to 14, by making two simplification choices. The first simplification was to focus on reproducing a unidirectional flow, so we kept only features describing the forward direction of the flow. The second simplification was to keep a group of features that could not be computed from other features, to reduce redundancy. For example, we kept the features *number of packet* and *IAT Mean*, but excluded the *Duration* feature since it can be computed from the former two. Our packet generator is then able to generate traffic from feature vectors of size 14 (see Table 2).

Flow features set
Source IP address
Source port number
Destination IP address
Destination port number
Protocol
Source to destination packet count
Minimum packet length from Source
Maximum packet length from Source
Mean packet length from Source
Standard deviation packet length from Source
Forward IAT Min
Forward IAT Max
Forward IAT Mean
Forward IAT Std

Table 3: Restricted set of flow features for LTGen generation.

6.3 Generation

In the following, we detail the different components of our generation approach: how to generate the distribution of a given traffic feature? How to emulate the network? How to generate a real packet? How to capture the traffic for analysis (i.e., evaluate our solution)?

6.3.1 Distribution generation

From the list of features in Table 3, the *Packet length* and *IAT* features have four parameters describing their distribution inside a flow: Min value, Max value, Mean value and Standard deviation. There are several distributions that can respect the four parameters (Min, Max, Mean, Std), we choose to use the *beta distribution*, in order to find one.

"The *beta distribution*[1] is a family of continuous probability distributions defined on the interval [0, 1] parameterized by two positive shape parameters, denoted by α and β".

In Figure 29, the beta distribution shape flexibility is demonstrated by plotting the probability density function under different α and β values.

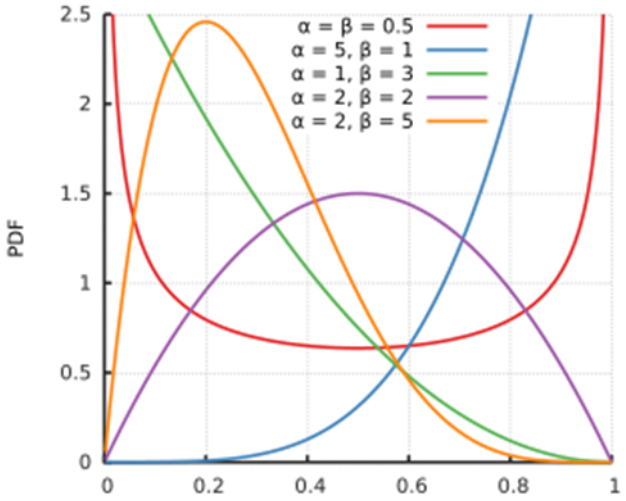

Figure 29: Beta distribution probability density function

From the four parameters (Min, Max, Mean, Std), we can compute α, β and a range to obtain a beta distribution that will allow us to generate values that will respect the four input parameters.

The main advantage of this solution is that from our configuration (Min, Max, Mean, Std), we obtain a distribution from which we can sample as many values as we want. This allows a great flexibility in the generated flow size, but also a good diversity since each value is randomly sampled. These properties (similarity, scale, diversity) ensure that the generated traffic, although synthetic, is realistic, i.e., close enough to the real network traffic.

6.3.2 Network emulation

Our generated traffic needs to be flown within a network testbed in which the IDS under test would be deployed. We have experimented with several tools to emulate a network where we would test our packet generation models. For each solution, we have studied the capabilities and limitations of the tool. We finally selected *Containernet [35]*, since it was the most versatile and practical tool to emulate a network.

Containernet is a fork of the project *Mininet Emulator [36].* Like *Mininet*, it can emulate a network of hosts, links and switches. It has an interactive Command Line Interface to interact with the hosts or to modify the network configuration. It can emulate a link between hosts with a specific delay, jitter, bandwidth and packet loss. The link characteristics are emulated using *NetEm [37]*. It runs on a single system which can be inconvenient with resource limitation when emulating large networks. On top of the *Mininet* functionalities, *Containernet* can emulate hosts from a chosen Docker image, which allows more flexibility in the network creation.

[1] https://en.wikipedia.org/wiki/Beta_distribution

6.3.3 Packet generation

First we tried to generate and send packets on the fly with a *Python* network packet crafting library, *Scapy*[2], but it was not possible to reach low inter arrival times values since *Scapy* was already introducing too much delay in the execution. Thus we opted to generate packets with *Scapy* beforehand, and put them inside a pcap capture file. The packets were consequently flown on the emulated network using *tcpreplay*[3] on the generated pcap file.

6.3.4 Packet capture and analysis

For analyzing a packet capture file, we used a *Tshark*[4] script to extract packet information into a csv file, followed by a *Python* script to compute flow features. This allows us to evaluate how similar our generated traffic is to the input features.

If the pcap file contains several flows, we first use we first have to use *Splitcap*[5], to extract flows into individual pcap files.

6.4 Experiments

In the following experiments, we used a simple network emulation with *Containernet*, with a sender machine and a receiver machine, linked by a switch with two links of 100 ms of delay and 1000 bits/s of bandwidth. We choose not to add any jitter in the links, in order to observe the impact of *Containernet* on the flow IAT features.

6.4.1 Experiment 1

The aim of the experiment is to reproduce a targeted profile inside our network emulation. We set two capturing points using *tcpdump*, one on the interface of the Sender machine, and the other one on the interface of the Receiver machine. The Sender machine will generate packets using *tcpreplay* from a pregenerated pcap file. We choose the UDP protocol since it is connectionless, and we fill up the payload randomly following the size given by the beta distribution.

A pcap will be generated beforehand: for each packet, a value for the *packet length* and the *IAT* will be sampled from their respective beta distribution.

The resulting profiles are displayed in Table 4.

On the Sender interface, we can observe that the packet generation from beta distribution with *tcpreplay* is quite similar to the targeted profile for *packet length* and *IAT* features. But we can see that the border values (Min, Max) are not always respected since from the beta distribution generated, they may be very unlikely to be produced. That is why the *Maximum packet length from Source* of the Sender interface is lower than the targeted one.

On the Receiver interface, we observe the same characteristics for *packet length*, but some fluctuations for the *IAT* feature. The mean value is respected, however IAT Min and IAT Max are quite different, IAT Max being one hundred times bigger for the Receiver interface. Nonetheless, Figure 31 shows that the global shape of the distribution remains the same.

[2] https://pypi.org/project/scapy/

[3] https://tcpreplay.appneta.com/

[4] https://tshark.dev/

[5] https://www.netresec.com/?page=SplitCap

Profile	Targeted profile	Sender interface flow profile	Receiver interface flow profile
Protocol	UDP	UDP	UDP
Source to destination packet count	10000	10000	10000
Minimum packet length from Source	10	10	10
Maximum packet length from Source	230	196	196
Mean packet length from Source	64	64.20	64.20
Standard deviation packet length from Source	32	32.03	32.03
Forward IAT Min	0.006	0.00660	0.0030
Forward IAT Max	0.02	0.0186	1.574
Forward IAT Mean	0.012	0.01203	0.01203
Forward IAT Std	0.002	0.00202	0.0201

Table 4: Flow profiles according to the location of capture

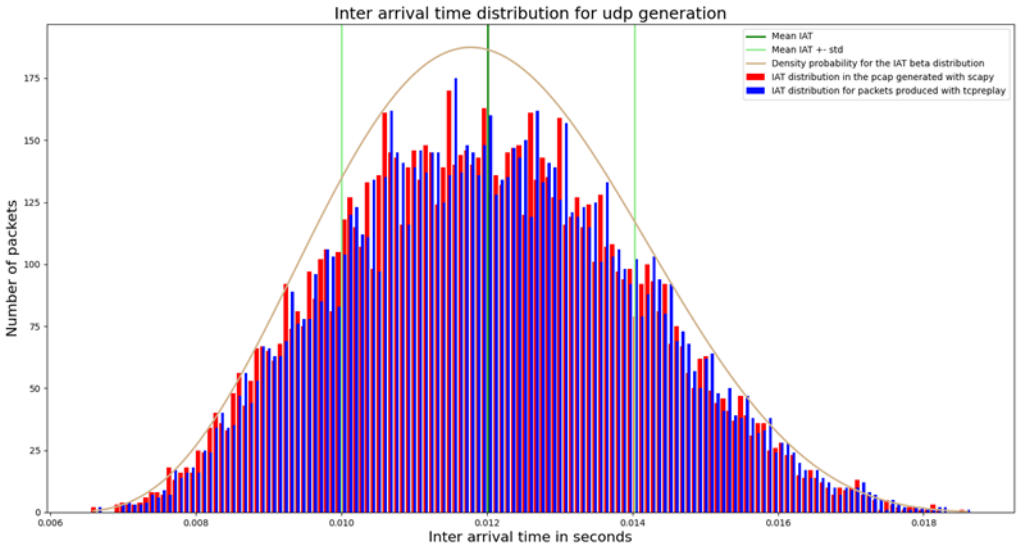

Figure 30: IAT of packets from the Sender interface

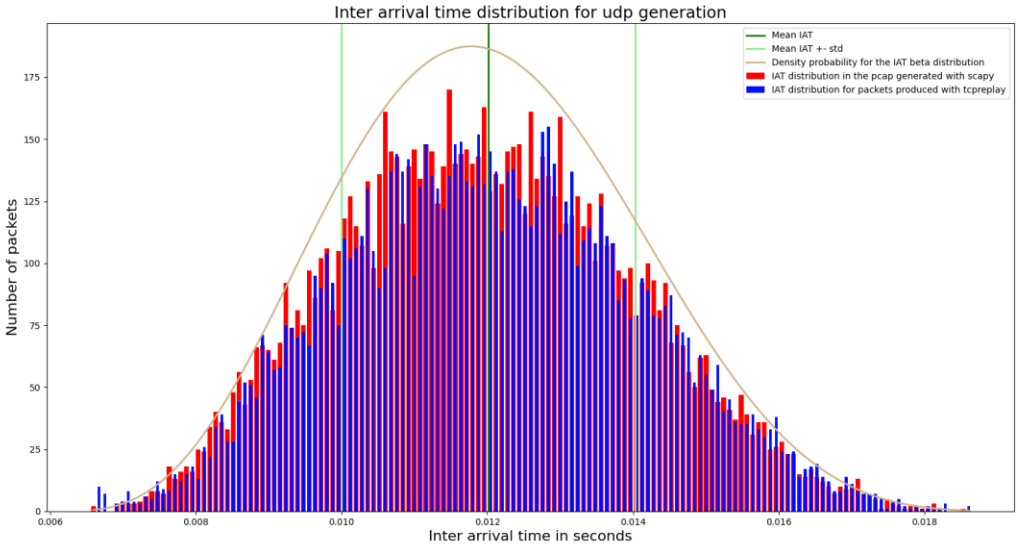

Figure 31: IAT of packets from the Receiver interface

Figure 30 and Figure 31 present two charts that group packets by inter arrival time (IAT). In *red*, packets generated by sampling the beta distributions, configured from the targeted profile, forming the pregenerated pcap. In *blue*, packets captured on an interface after being sent by tcpreplay from the pregenerated pcap. In Figure 30, is represented the capture at the Sender interface, while in Figure 31, it is from the Receiver interface.

In *dark green*, is represented the targeted Mean IAT value. In *light green*, is represented the Mean (more or less the Standard deviation of IAT). It defines an interval where 68% of the generated values should be.

In *brown*, is represented the density probability of the beta distribution configured from the targeted profile (Min, Max, Mean, Std). It displays a global pattern of value distribution. we can see that, in this profile configuration, border values are quite unlikely to be sampled.

6.4.2 Experiment 2

This second experiment aims to test a different profile configuration. More precisely, the case where: $Mean - Std < Min$. We will focus on the *IAT* feature and the beta distribution. Table 4 presents the results of the generation of IAT.

In Figure 32, we observe that the Beta distribution can adapt from this situation by producing a distribution that looks like a Pareto distribution.

6.5 Discussions

6.5.1 Standard error

The deviation of a generated profile to the targeted profile depends on the number of packets to be generated. The standard error is defined as the ratio of the standard deviation on the square root of n, the number of samples. Thus, for flows with a small number of packets and a high standard deviation, the standard error of a generated profile to the targeted profile will be high.

Profile	Targeted profile	Sender interface flow profile
Protocol	UDP	UDP
Source to destination packet count	10000	10000
Forward IAT Min	0.0001	0.000104
Forward IAT Max	0.6	0.048304
Forward IAT Mean	0.002	0.001929
Forward IAT Std	0.004	0.003929

Table 5: Flow profiles for the specific case $Mean - Std < Min$

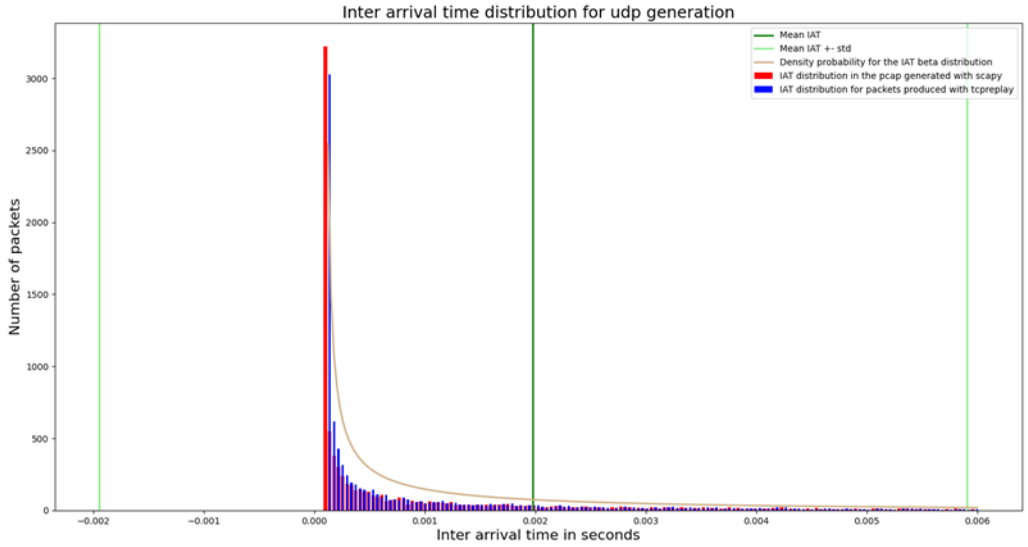

Figure 32: IAT Distribution for the specific case $Mean - Std < Min$

In the CICIDS2017 dataset, we observed a wide majority of flows containing less than 20 packets (more than 80% of the database). So this generation method may not be relevant for a majority of flows, since we will generate flows with a high standard error, in other words flows with a different profile.

6.5.2 Temporal consistency

For each packet of a generated flow, the packet length and IAT attributes are sampled independently. Yet, the value of these attributes may be correlated with the characteristics of the previous packets. But in order to improve the temporal consistency between packets of a flow, new features have to be designed to characterize it.

Chapter 7 List of Abbreviations

Abbreviation	Translation
AES	Advanced Encryption Standard
CPU	Central Processing Unit
HPC	High Performance Counters
KPTI	Kernel page-table isolation
L1	First-Level Cache
LLC	Lowest-Level Cache
PMC	Performance Monitor Counters
PMI	Performance Monitoring Interrupt
PMU	Performance Monitoring Unit
SR	Software Requirement
TLB	Translation Lookaside Buffer

Chapter 8 Bibliography

[1] SPARTA D5.2 Demonstrators specifications. January 2021.

[2] SPARTA D5.3 Demonstrator prototypes. January 2021.

[3] SPARTA D5.4 Integration on demonstration cases and validation. January 2022.

[4] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 1: Introduction and general model.

[5] Common Criteria for Information Technology Security Evaluation, Version 3.1, revision 5, April 2017. Part 3: Assurance security components.

[6] https://satra.iit.cnr.it/

[7] https://satra.iit.cnr.it/api/v1/

[8] https://satra.iit.cnr.it/token.jsp

[9] https://git.shibboleth.net/view/

[10] Alessandro Pellegrini. 2014. Techniques for Transparent Parallelization of Discrete Event Simulation Models. Ph.D. Dissertation. Sapienza, University of Rome. http://www.dis.uniroma1.it/{~}pellegrini/pub/pellegrini-phd.pdf

[11] AMD. 2019. Processor Programming Reference (PPR) for AMD Family 17h 01h,08h, Revision B2 Processors.

[12] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-Timing Template Attacks. In Advances in Cryptology, Mitsuru Matsui (Ed.). Lecture Notes in Computer Science, Vol. 5912. Springer, Berlin, Heidelberg, 667–684. https://doi.org/10.1007/978-3-642-10366-7_39

[13] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A program supervision framework. In Electronic Notes in Theoretical Computer Science, Vol. 89. 47–69. https://doi.org/10.1016/S1571-0661(04)81042-9

[14] Michael Larabel and Matthew Tippett. 2011. Phoronix Test Suite.

[15] Colin Ian King. 2017. Stress-ng: A stress-testing Swiss army knife. (2017)

[16] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping Bits in Memory without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In Proceedings of the 41st International Symposium on Computer Architecture (ISCA). IEEE, 361–372. https://doi.org/10.1109/ISCA.2014.6853210

[17] https://github.com/vusec/revanc

[18] https://github.com/vusec/xlate

[19] https://github.com/paboldin/meltdown-exploit

[20] https://github.com/Eugnis/spectre-attack

[21] Marco Spaziani Brunella, Giuseppe Bianchi, Sara Turco, Francesco Quaglia, and Nicola Blefari-Melazzi. 2019. Foreshadow-VMM: Feasibility and Network Perspective. In Proceedings of the 2019 Conference on Network Softwarization (NetSoft). IEEE, 257–259. https://doi.org/10.1109/NETSOFT.2019.8806712

[22] Nate Lawson. 2009. Side-Channel Attacks on Cryptographic Software. IEEE Security & Privacy Magazine 7, 6 (nov 2009), 65–68. https://doi.org/10.1109/MSP.2009.165

[23] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – And its application to AES. In

Proceedings of the 2015 Symposium on Security and Privacy. https://doi.org/10.1109/SP.2015.42

[24] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In Proceedings of the 2015 Symposium on Security and Privacy. IEEE, 605–622. https://doi.org/10.1109/SP.2015.43

[25] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A timer-free high-precision L3 cache attack using intel TSX. In Proceedings of the 26th USENIX Security Symposium.

[26] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Proceedings of the 23rd USENIX Security Symposium. 719–732.

[27] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In Lecture Notes in Computer Science, Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez (Eds.). LNCS, Vol. 9721. 279–299. https://doi.org/10.1007/978-3-319-40667-1_14 arXiv:1511.04594

[28] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious Management Unit: Why Stopping Cache Attacks in Software is Harder than you Think. In Proceedings of the 27th USENIX Security Symposium.

[29] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. FORESHADOW: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In Proceedings of the 27th USENIX Security Symposium

[30] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. Proceedings of the 27th USENIX Security Symposium (2018).

[31] Koroniotis N, Moustafa N, Sitnikova E, Turnbull B. Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset. Future Generation Computer Systems. 2019. pp. 779–796. doi:10.1016/j.future.2019.05.041

[32] Moustafa N, Slay J. UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). 2015 Military Communications and Information Systems Conference (MilCIS). 2015. doi:10.1109/milcis.2015.7348942

[33] Moustafa N. A new distributed architecture for evaluating AI-based security systems at the edge: Network TON_IoT datasets. Sustainable Cities and Society. 2021. p. 102994. doi:10.1016/j.scs.2021.102994

[34] Sharafaldin I, Lashkari AH, Ghorbani AA. Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization. Proceedings of the 4th International Conference on Information Systems Security and Privacy. 2018. doi:10.5220/0006639801080116

[35] Peuster M, Karl H, van Rossem S. MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments. 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). 2016. doi:10.1109/nfv-sdn.2016.7919490

[36] Lantz B, Heller B, McKeown N. A network in a laptop. Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks - Hotnets '10. 2010. doi:10.1145/1868447.1868466

[37] Jurgelionis A, Laulajainen J-P, Hirvonen M, Wang AI. An Empirical Study of NetEm Network Emulation Functionalities. 2011 Proceedings of 20th International Conference on Computer Communications and Networks (ICCCN). 2011. doi:10.1109/icccn.2011.6005933