Erno Pasanen

COMPARING AFL SCALABILITY IN VIRTUAL- AND NATIVE ENVIRONMENT

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2020

TIIVISTELMÄ

Pasanen, Erno
Comparing AFL scalability in virtual- and native environment
Jyväskylä: Jyväskylän yliopisto, 2020, 96 s
Tietojärjestelmätiede (Kyberturvallisuus), pro gradu -tutkielma
Ohjaaja: Costin, Andrei

Tämän työn lähtökohtana oli tutkia automaattisien haavoittuvuusetsintä työkalujen (fuzzereiden) skaalautuvuutta natiivissa- ja virtuaalisessa suoritusympäristössä. Tutkielma suoritettiin monitapaustutkimuksena, jossa analyysi yksikkönä toimi fuzzeri American Fuzzy Lop (AFL). Monitapaustutkimuksen tavoitteena oli millaisella konfiguraatiolla AFL toimii parhaiten, hidastaako suoritusympäristö fuzzeria ja skaalautuuko AFL rinnakkaisajossa olemassa olevan teorian mukaan?

Kirjallisuuskatsauksen perusteella tunnistettiin neljä mittaria: koodin kattavuus, löydettyjen bugien määrä, suoritusten määrä sekunnissa, sekä yhteisen bugin löytämiseen kulunut aika. Monitapaustutkimus jaettiin viiteen osaan, joista jokainen osa toistettiin natiivissa ja virtuaaliympäristössä. Osien toistaminen suoritettiin skriptaamalla.

Instanssien ajaminen ei sujunut täysin odotuksien mukaisesti. Pilotti tapauksen ajaminen vaati kaksi uusintakertaa, sillä tuloksien perusteella oli nähtävissä, että AFL ei ollut käynnistynyt oikein. Tuloksia kerätessä huomattiin myös, että yhden päivän koe oli uusittava virheen vuoksi. Lisäksi yhden virtuaalisen tapauksen tuloksia ei ollut kirjattu tuntemattomasta syystä, mutta tämä ei ollut este tutkimuksen tuloksien analysoimiseksi.

Tutkimuksen tulokset analysoitiin Mann-Whitney U-testillä sekä Vargha-Delaney $\hat{A}_{12}$ vaikutuksen suuruus testillä. Koodin kattavuutta ei voitu arvioida, sillä tulokset olivat liian homogeenisia. Löydettyjä bugeja oli yhteensä seitsemän, mutta bugeja oli löydetty hyvin harvakseltaan, jolloin vertailua ei voitu suorittaa. Suoritusnopeuden tapauksessa konfiguraatioiden keskiarvon mittaaminen muodostui ongelmalliseksi, koska keskiarvo suoritusnopeudesta konvergoitui renkien (slave) tuloksien ympärille. Suoritusnopeutta mitattiin täten laskemalla fuzzereiden yhteenlaskettu keskiarvo. Kumulatiivisen keskiarvon lisäksi kaikista tapauksista löytyi yhteinen bugi (read_utmp) jota voitiin käyttää tehokkuuden mittaamiseen.

Tuloksien perusteella voidaan todeta, että käyttäessä monta isäntää fuzzeri nopeutuu, mutta lisättäessä renkejä sen kyky löytää bugeja paranee. Vastaavasti virtuaali- ja natiivi toteutus eivät tehollisesti eronneet toisistaan merkittävästi. Lopuksi voidaan todeta, että fuzzaaminen skaalautuu erittäin tehokkaasti käyttäessä kahta tai kolmea tietokoneen ydintä.

Asiasanat: haavoittuvuus, haavoittuvuuksien etsintä, fuzzerit, monitapaustutkimus, skaalautuvuus, tehokkuus

ABSTRACT

Pasanen, Erno
Comparing AFL scalability in virtual- and native environment
Jyväskylä: University of Jyväskylä, 2020, 96 p.
Information Systems (Cybersecurity), Master's Thesis
Supervisor: Costin, Andrei

Object of this study is to explore scalability of automatic vulnerability discovery tools (Fuzzers) in virtual and native execution environments. Multiple-case study was executed while the unit of analysis within was fuzzer American Fuzzy Lop (AFL). Research questions for this multiple-case study were: Does AFL scale according to known theoretical models, how is the scalability hindered through virtualization and how does the performance differ when different AFL configurations are used?

From current academia four different metrics were identified: code coverage, bug count, execution speed and time to find shared bug. Multiple-case study was done through five cases in both native and virtual environment. Execution of cases was done through scripting.

Execution of cases had few problems. Pilot study had to be repeated twice because of irregularities in data showing that AFL had not started properly. During gathering the results, it was discovered that one day worth of data had to be rerun. In addition, for unknown reason, one virtual instance run is forever lost, but it does not hamper the analysis of this study.

This study used Mann-Whitney U-test and Vargha-Delaney $\hat{A}_{12}$ effect size measurement to assess metrics. Code coverage proved to be homogenous and was therefore discarded. Instances found a total of seven unique bugs and therefore results were too sparse to be analyzed. Execution speed proved to be biased as the averages of instances skewed towards larger dataset of slave configured fuzzers. Therefore, cumulative values of execution speed per configuration were used as metric. Furthermore, a single shared bug was found (read_utmp) which could be used to assess performance.

Study concludes that configuration of instances favors execution speed while masters are used, while bug discovery is enhanced by using slave configured instances in addition to masters. No significant performance difference was found between virtual and native environments. Finally, it can be said that fuzzers scale well in two and three core instances.

Keywords: vulnerability, vulnerability discovery, fuzzers, multiple-case study, scalability, efficiency

FIGURES

TABLES

EQUATIONS

INDEX

1 Introduction

1.1 Motivation

"Is it better to try and compromise with efficient approaches that balance coverage and speed, or to use a tedious but highly productive approach and then throw lots of cheap computing power at it?" - Danny Bradbury, Naked security article "Faster fuzzing ferrets out 42 fresh zero-day flaws" (Bradbury, 2018)

Software is under attack. This conflict has two sides: those that continuously try to secure the software or those that try to exploit it for their own means (Microsoft, 2006). Both sides have built their respective tools and processes in order to counter the other. Software defenders have their secure software development cycle (see e.g. Microsoft SDL) (Microsoft, 2019)and testing tools and approaches: for example ISO/IEC/IEEE 29119 part 4: test techniques depicts multiple tools and approaches (ISO/IEC/IEEE, 2015). Securing of software is done both in-house development and in production environment -by other than developers.

1.1.1 One angle of attack for software

The attacker can use the same tools and processes to find gaps in security and leverage them in order to gain (for example) unauthorized access system. Attacker is not however limited to same toolset as the defender and can in many cases use different and more advanced tools than the defender. Attacker is not limited attacking only the software as many other techniques are available: phishing, social engineering, and supply chain attacks are all examples of attack vectors that can aid the attacker. (McClure, Scambray, & Kurtz, 2012, pp. 314-322)

One of these fore-mentioned methods is fuzzing. ISO/IEC/IEEE classifies fuzzing as "mathematical based testing". In this type of testing the input and output descriptions are mature enough that automated testing planning input generation, and test cases can be created. Different inputs are randomized (Random test case generation) and combined (Combinatorial testing), and their test coverage is measured statistically. (ISO/IEC/IEEE, 2013, p. 32)

Fuzz-testing is a vast method ranging from pure cloud solutions to workstation level testing and from commercial- and open source products wrapped in virtual containers to native operating system environment specific applications. Examples include but are not limited to:

- Cloud based Microsoft Security Risk Detection (MSRD) (Linn, 2017),
- Google´s OSS-fuzz (Google, 2019) a cloud based open source fuzzer,
- containerized OUSPG´s Cloudfuzzer (University of Oulu, 2017)
- Linux based American Fuzzy Lop (AFL) (Zalewski, American Fuzzy Lop (2.52b), 2016), a workstation based fuzzing software,
- WINAFL (Fratric, 2016), an windows branch of AFL.

1.1.2 Differences between online and offline software

Having a system or service established online induces the problem of data confidentiality. Regulatory bodies of Finland have published VAHTI-instruction 3/2012 ("Valtionhallinnon tieto- ja kyberturvallisuuden ohjausryhmän", Governmental information-, and cybersecurity control group) and KATAKRI 2015 (Information security audit tool for authorities) to establish a base level for online (governmental) services. VAHTI instructions allow the service to be either in-house or bought, while considering the confidentiality and availability of the system.

In all cases where said system is not governmentally controlled (Dedicated environment, government owned shared platform, or bought platform of which both personnel and equipment can be audited), or regulated, the information that can be held in said system is considered "public" - not confidential (Ministry of Finance Finland, 2013). KATAKRI incorporates many different resources in order to have distinct definition for public and restricted environment requirements. It is a tool for (national) authorities to assess organizations ability to protect confidential information (Finnish Ministry of Defence, 2015).

Finding software vulnerabilities is confidential: both sides of the "battle" gain advantage on having proprietary information on tested program. Therefore, having the software tested on online environment might not be beneficiary for the defender. Having an offline (and segregated) development and testing environment builds yet another barrier for the attacker.

1.1.3 Focusing this study

In this research, focus is an offline approach to fuzzing. Fuzzing on host-based machine can be just as effective as massive cloud instances (Pham, Böhme, Santosa, Roychoudhury, & Câciulescu, 2018). Most prevalent advances of fuzzing do not come from adding more hardware to handle the task, rather optimizing the individual components of fuzzer. Example of this element optimization would be adding a input generating element to AFL (Peach support) that helps the AFL craft smarter and more valid inputs to program (“Virtual file format chunks”) (Pham, Böhme, Andrew, Căciulescu, & Roychoudhury, 2019).

Offline fuzzing has its downsides: the hardware is not scalable, whereas in online (cloud) environment it is. Therefore, the offline approach for fuzzing might benefit of certain optimization. Optimizations in any part of a fuzzer raises a question of did the optimization raise overall efficiency of fuzzer, or did it incur more overhead rendering done optimization useless?

When measuring efficiency of an approach it is interesting to ask a question ”is this the most effective approach”? Intuitively, the more you assign resources at a task the faster the task is done. Modern fuzzers are able to regulate internal processes in order to be more effective, for example AFLFAST (Böhme, Pham, & Roychoudhury, 2017), development that is apparent in AFL from its creation to present (Zalewski, AFL-changelog, 2017). Therefore, are fuzzers effective when ran in parallel with the same target? Are there limits that are not apparent when using parallel execution? Are there theoretical mores that could predict the scalability off fuzzers? – These were the starting sparks for this study.

1.2 Research goals and starting point

This chapter serves as an outline for research questions and the sub questions that must be answered before main questions are validated. Questions are derived from Chapter 1.1 and improved during literacy study (Chapter 2). There is abundance of research considering fuzzers (see Chapter 1.2.2), but it mainly focuses on how research is done and that the subjects are mainly fuzzers compared to each other on efficiency of bug discovery.

1.2.1 Research questions

Main questions:

1. Does AFL scale according to any known theoretical scaling model?
2. How is the scalability hindered through VM-virtualization?
3. How does the performance differ when using a different number of master and slave configurations in AFL?

Sub questions

1. How do you measure AFL scalability?
2. What are the metrics in performance measuring of AFL?

3. Do minor changes in the test setup change performance metrics?

1.2.2 Previous studies

AFL is a fuzzer, that has been first released in 12th of November in 2013 by its creator Michal Zalewski, most current version of AFL is 2.52b (released 4.11.2017) (Zalewski, AFL-changelog, 2017). AFL development is community driven, for example "AFL-_FAST_CAL", which is a power management function used in AFL-Fast (Böhme, Pham, & Roychoudhury, 2017) was added in 19th August in 2017. The most important articles that are referred in this study to define AFL and its processes are Hongliang & al. "Fuzzing: State of the Art" (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018), Klees & al. "Evaluating Fuzz testing" (Klees, Ruef, Cooper, Wei, & Hicks, 2018), and Pham & al. "Smart Greybox Fuzzing" (Pham, Böhme, Santosa, Roychoudhury, & Câciulescu, 2018).

Hongliang & al. have gathered and analyzed 171 articles in IEEE Xplore, ACM Digital Library, Springer Online Library, Wiley InterScience, USENIX and El-sevier Sciencedirect Online Library using phrases like fuzz testing, fuzzing, fuzzer, random testing and swarm testing. This approach has led to a comprehensive article about the current process and state of fuzzers, including AFL. (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018)

Klees & al. have surveyed 32 articles regarding AFL experimental evaluations. The key finding in this article is that amongst surveyed articles several discrepancies in either the empirical demonstration or taking account the random nature of fuzzing. Klees & al. propose an algorithm for empirical evidence gathering and statistical methods to counter the random nature of fuzz testing. (Klees, Ruef, Cooper, Wei, & Hicks, 2018)

Pham & al. reviewed the process of input file generation when the file format has more complicated structure. This extended AFL's input functionality to include virtual file format chunks, power schedule, and mutational functionality. This resulted in performance gain over the standard AFL and called this extended AFL as "AFLSmart". Article incorporated a case study that compared AFL with AFLSmart providing methodology that is used in this study. (Pham, Böhme, Santosa, Roychoudhury, & Câciulescu, 2018)

The theory on general fuzzer schema for this research is derived from "Fuzzing for Software Security Testing and Quality Assurance" by Ari Takanen et al. (Takanen, Demott, & Miller, 2008). The theory on fuzzers is complimented by several articles by other authors for example Marcel Böhme & al "Coverage-based Greybox Fuzzing as Markov Chain" (Böhme, Pham, & Roychoudhury, 2017), Junjie Wang & al. "Skyfire: Data-Driven Seed Generation for Fuzzing" (Wang, Chen, Wei, & Liu, 2017) and Sanjay Rawat & al. "VUzzer: Application-aware Evolutionary Fuzzing" (Rawat, et al., 2017).

Theory of performance testing derives from McCool et al. "Structured parallel programming: patterns for efficient computation". Practical considerations for evaluating fuzz testing have been revised through George Klees et al. "Evaluating fuzz testing" (Klees, Ruef, Cooper, Wei, & Hicks, 2018), Jian Liang & al

"Fuzz Testing in Practice: Obstacles and Solutions" (Liang, Wang, Chen, Jiang, & Zhang, 2018) and Hongliang Liang & al. "Fuzzing: State of Art" (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018).

Starting point for philosophy and the crafting of method is defined through Saunders & al. "Research onion" (Saunders, Lewis, & Thornhill, 2012, p. 128). Methodology was further developed through Robert K. Yin "Case Study Research Design and Methods". Both of these methodology books are highly rated and referenced in Google Scholar (Saunders: over 20 000 references (Google, 2019).Yin: over 16000 references (Google, 2019).

1.3 Proposed scientific methodology

In this chapter the scientific methodology is deduced by using Saunders & al. "*Research methods for business students*" (Saunders, Lewis, & Thornhill, 2012, p. 128). From previous studies it is apparent (see previous chapter) that experiment must be done as fuzzers are essentially randomized algorithms that are tasked to find bugs (Miller, Fredriksen, & So, In Proceedings of the Workshop of Parallel and Distributed Debugging, 1990). Multiple-case study is chosen as the framework for this study and data collection and analysis procedures are outlined from this framework.

1.3.1 Research philosophy

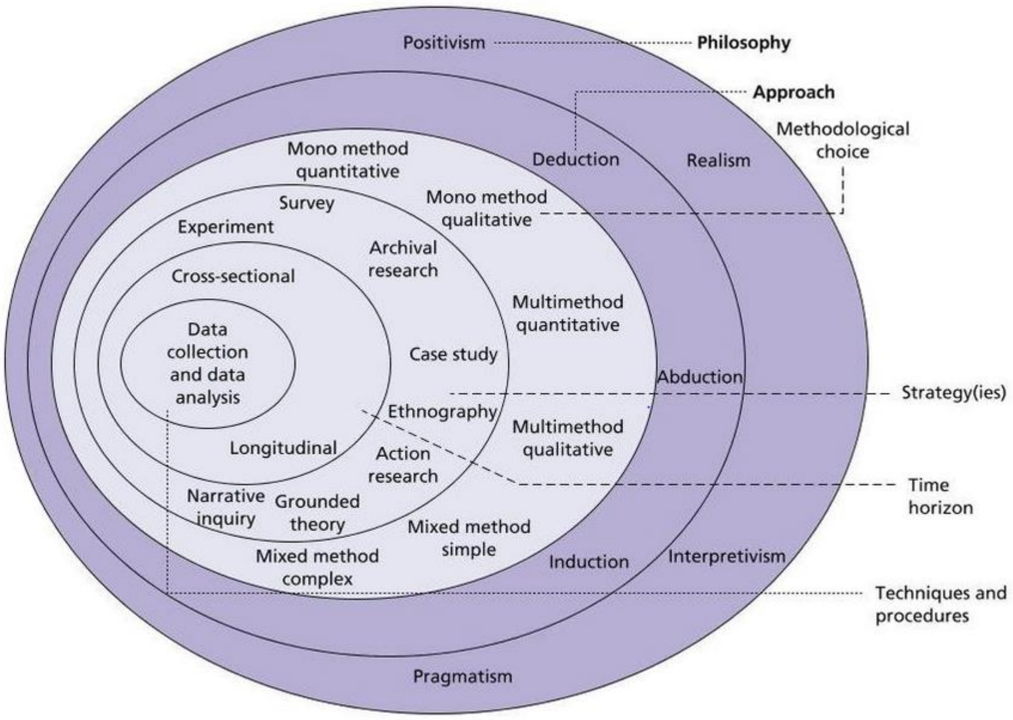


Figure 1 Research onion according to Saunders & al.

The philosophy of science relies on series of assumptions that form the respective understanding of the phenomena that is under investigation while simultaneously affecting chosen methodology (Saunders, Lewis, & Thornhill, 2012, p. 128). These assumptions contribute to the philosophy, approach, strategy, and techniques and procedures used in scientific research and are summarized in Figure 1, above, which on outer (darker) layer depicts the available philosophy and approach and in the inner layer (lighter) comprises of available methodologies, strategies, and techniques and procedures.

Phenomena that is under scrutiny here (scalability of software component) presumably provides data that can be seen and interpreted. An Assumption in this case would be that AFL provides data that has causality, in other words: having more fuzzers working on the same problem yields results faster. This kind of assumption leads to positivistic philosophical approach where only observable results lead to credible data and might credit (or discredit)to an existing theory (Saunders, Lewis, & Thornhill, 2012, pp. 134-145). Realism would state that there is no causality between the human and program but as the whole information system is an object of human ingenuity this cannot be the case. The data of AFL is not open for interpretation as it is (in this research) quantitative in nature leaving little room for interpretative approach.

Approach of this research is deductive in nature: assumption is that AFL´s performance enhances as more and more instances are used but it is unknown how the systems performance scales comparing to single instance. As we are working reducing data to a known theoretical framework, inductive approach is not used. Abductive approach is also not viable, as it requires the combination of both inductive and deductive approaches. (Saunders, Lewis, & Thornhill, 2012, pp. 145-147)

1.3.2 Chosen methodology and research strategy

AFL provides its data in form on exported text file and graph (Zalewski, AFL Readme, 2017). There are two kinds of data: runtime data (referred as status screen) and exported runtime data. This research focuses on the exported data, as it is the concentrated output of AFL. This data is quantitative in nature as it takes form of numbers (Saunders, Lewis, & Thornhill, 2012, p. 161).

Presumed quantitative approach does not exclusively deter qualitative approach but it could be hard to make a qualitative study and exhaust all the possibilities that factor into scalability of a computer. Hongliang & al. have used qualitative approach (literacy review) to scope the current state of fuzzers but does not yield information on how scalable fuzzers are (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018). Quantitative data (in this case, number of crashes) has been used to compare different fuzzers (Klees, Ruef, Cooper, Wei, & Hicks, 2018). Comparing different fuzzers does not reveal scalability information regarding single fuzzer but the ranking of fuzzers in a controlled environment (Klees, Ruef, Cooper, Wei, & Hicks, 2018). Both Klees & al. and Hongliang & al. have made tremendous effort to compare different fuzzers in order to form a consensus on roadmap for future fuzzers, but they do not touch the subject of fuzzer scalability. In this research quantitative approach is taken as it studies differences and causality of numbers and is usually paired with deductive positivistic approach (Saunders, Lewis, & Thornhill, 2012, pp. 162-163); therefore, we can presume that this methodology yields results.

In this study, the research strategy derives from chosen philosophical approach and methodology. In most quantitative studies either experimental or survey method is used (Saunders, Lewis, & Thornhill, 2012, p. 163). Experimental method relies on causality between two variables, called independent variable and dependent variable (Saunders, Lewis, & Thornhill, 2012, p. 174). As we are using a program that relies on automated test case generation (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202) that is deterministic on start and randomized the further AFL is executed (Zalewski, AFL technical details, 2017) it could be hard to establish independent variable that produces the same result every time it is used. This independent variable is produced by statistical means observing the behavior and outputted data of AFL.

When exploring real world phenomena in its context a case study is a valid option for research strategy. Case studies divide in two dimensions: single case versus multiple cases, and holistic case versus embedded case (Saunders, Lewis, & Thornhill, 2012, pp. 179-181). Key points dividing these dimensions are context and Unit of analysis. The types of designs for these kinds of studies are pictured in Figure 2, which shows that case study can be divided by single- and multiple-case design, and its approach of holistic or embedded. A single case scenario considers only a single critical or unique case, and as we need information on native- and virtual environments: single case approach is discarded. Multiple-case approach is better suited for this research as we can presume that results in different cases are causally linked to one another by having the same restrictions and structure (also known as construct) through case study methodology (Yin, 2014, pp. 45-49) .

Holistic multiple-case study would require the results to be pooled within context of study and not examined as different cases (Yin, 2014, p. 62). When taking into account the unit being analyzed (AFL), embedded case study approach is more favorable than holistic approach, as it explores the interdependencies of units within same case. These interdependencies within the case structure and context presumably yield results on comparison of virtual and native environments.

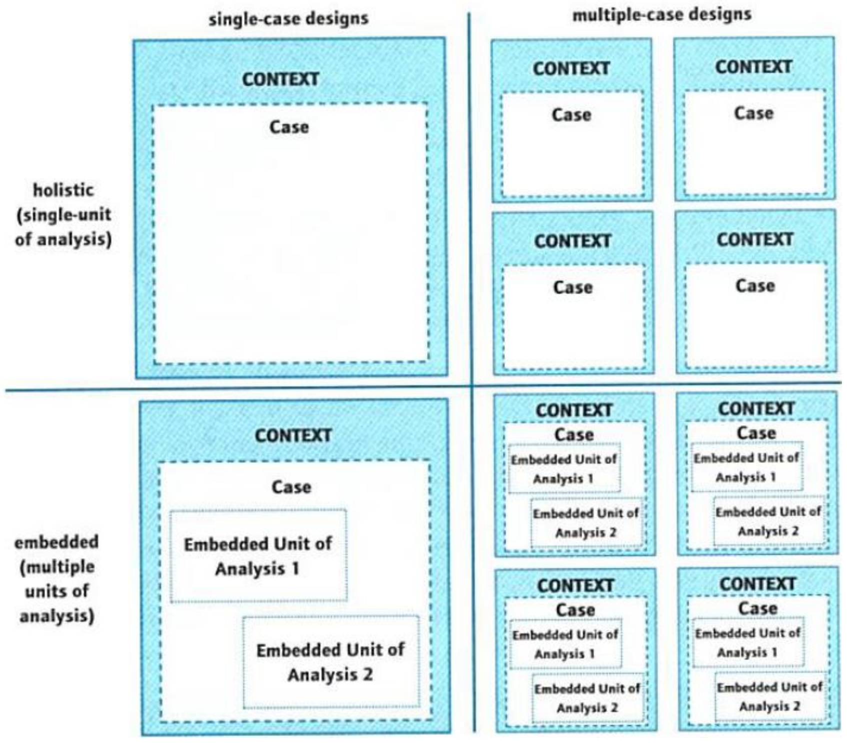

single-case designs
multiple-case designs
holistic (single-unit of analysis)
embedded (multiple units of analysis)
CONTEXT
Case
Embedded Unit of Analysis 1
Embedded Unit of Analysis 2

Figure 2 Types of Designs for Case Studies according to Yin (Yin, 2014)

Timeframe of this study is reflected through chosen research strategy and unit of analysis (case): process of this study is not a continuous process that would require continuous effort, rather the cases have a starting point and an endpoint. AFL is ran for predetermined time and then collected data is analyzed and compared to other cases in same context. Boundaries set to case study limit the timeframe to *cross-sectional study* (Saunders, Lewis, & Thornhill, 2012, p. 190) as it observers a phenomena (*performance*) in set amount of data points (*cases*).

1.3.3 Data collection in case studies

Data collection divides to two different phases: preparation, and collection of actual data. Preparation can be a process that is more complex than the process of study, as poor preparation can jeopardize the whole research. Preparation includes the following (Yin, 2014, p. 71):

1. mapping out the skillset needed for research
2. training for specific case study
3. refining the approach and process of case study (protocol)
4. deciding candidate cases
5. conducting a pilot study.

All of the above requirements reflect the familiarity on case study subject, and the ability to react to different emerging threats during and after data-collection.

Training has been undertaken before data collection on many different are of computer science from which applicable to this research have been derived from research questions. These are usage of AFL (RQ1 and RQ3), ability to use both native and virtual environments (RQ2), and ability to convert data into plotted graph (RQ1-3).

Process of case study divides into four elements (Yin, 2014, pp. 84-94): Overview, data collection procedures, using instrument to gather data, and documenting the results. Both overview and documentation is handled through thesis process, but process regarding data collection procedures and instruments are explained in Chapter 3.

One-phase approach for candidate screening is adopted (Yin, 2014, p. 95), as cases can be selected to accommodate the progression of performance and scalability through different configuration of AFL. Candidate cases in this study are numerous, as both native and virtual environments provide the different contexts with the same amount of analysis units within.

The role of pilot case study is to test and refine the proper procedure and collection methods (Yin, 2014, pp. 96-97). In this case the pilot case study is conducted to gain knowledge on single instance of AFL performance in both native- and virtual environments. As the operation system must be chosen in Linux domain, the results of pilot study have crucial role in refining the operating system of subsequent cases. Finally, pilot case results are used as the starting point for measuring scalability.

1.3.4 Case data analysis

> "The analysis of case study evidence is one of the least developed aspects of doing case studies" --Yin Robert K. on analyzing case study evidence in Case Study Research Design and Methods (Yin, 2014, p. 133)

General strategy of case data analysis links the case study data to the concept of study. These strategies combined with insight provide starting point and direction to analysis techniques. Yin proposes four general analytical strategies (Yin, 2014, pp. 136-142):

1. relying on theoretical propositions
2. working your data from the "ground up"
3. developing a case description
4. examining plausible rival explanations

Theoretical propositioning of this study is to compare performance metric differences between cases. In computers these metrics are speedup and efficiency. Speedup measures the ratio of one worker latency to multiple worker latency and efficiency measuring the use of available resources. (McCool, Robinson, & Reindeer, 2012, p. 56).

Case evidence (Yin, 2014, p. 133) is provided through AFL runtime data (Zalewski, AFL plot data, 2014) which states that following runtime data is saved:

Unix time, Cycles done, Current path, total paths, pending paths, pending favorite paths, size of bitmap, unique crashes, unique hangs, max depth (of coverage) and executions per second. According to Klees & al. most common dataset used is unique crashes (Klees, Ruef, Cooper, Wei, & Hicks, 2018) but as speedup is a function of work done and their span of critical path (McCool, Robinson, & Reindeer, 2012, p. 63) the time in which work is done remains a metric that must be incorporated. As the evidence and metrics do not yield explicit results together but most likely provide some patterns that are recognizable. Therefore, a strategy that includes working and manipulating data must be adopted.

Developing case descriptions and exploring rival plausible explanations as a strategy are strategies that are less likely to be employed in this study. When developing case descriptions, the vastness of data and its perceived structure form the case study restrictions and research which have been established at this point. As for regarding rival theories: Klees & al. (Klees, Ruef, Cooper, Wei, & Hicks, 2018) have established that rivalry in fuzzing academics focuses on comparing the results of fuzzers through statistical (or other) means, not studying their optimal use and scalability.

Analytic strategy of this study relies on data manipulation in order to find patterns, insights, or concepts that seem promising in performance metric context. According to Yin (Yin, 2014, p. 135) manipulation methods include but are not limited to array manipulation, categorizing said arrays and matrices, displaying data in different graphics (for example: plots and arrays, which are common in comparing different fuzzers to other fuzzers (Klees, Ruef, Cooper, Wei, & Hicks, 2018)), determining frequency of events, and using a temporal scheme to order data.

1.4 Conclusions

Software exploitation is a rival process that uses the same methods for both attacking and defending. These methods included automated testing, and automated input generation (fuzzing). Testing can be done both in scalable online environments and in offline hardware restricted environments. In this study offline environment is chosen through it having better confidentiality than online environments.

The current field of academia surrounding fuzzing is divided: most of the comparisons done in and before 2017 have been discredited through lacking scientific procedure, but advances have been made by introducing extensions to AFL. Fuzzing as a tool is still usable, despite being random in nature.

Philosophy of the research is positivistic in nature relying on data provided by the fuzzing suite AFL. This data is quantitative in nature and must be manipulated in order to gain theoretical validation to scalability of multiple workers. The manipulation is done within a case study framework, utilizing performance theory´s theoretical propositioning and emerging patterns within data.

2 Overview of related fields

This chapter uses literacy as an fountain of information to overview the related fields of fuzzing, performance theory and virtualization in their respective chapters. Fuzzing is overviewed through history to its current form as and tool for vulnerability researchers. Performance theory offers overview for the framework of metrics that must be constructed in order to measure *speedup* and *efficiency* of multi-worker instances. Virtualization is commonplace, but overview of virtualization layers and technologies yields information on how the experimental set-up if this multiple-case study can be constructed.

2.1 Fuzzing

Fuzzing is a software testing method that explores programs different code paths and provides information on what inputs potentially crash or causes the program to fail (Takanen, Demott, & Miller, 2008, p. 24).

2.1.1 Brief history

Fuzzing was proposed as an idea by Miller et al. in 1989 (Miller, Fredriksen, & So, In Proceedings of the Workshop of Parallel and Distributed Debugging, 1990) together with a tool called "fuzz" which generated random stream of input characters to a target program. The test results showed that most UNIX programs did not handle random inputs adequately. Both Hongliang et al. (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018)and Takanen et al. (Takanen, Demott, & Miller, 2008, pp. 22-23)perceive this as the first instance of successful fuzzing. Takanen et al. also denote that program called "Monkey" was used in Quality Assurance (QA) field to test UI components in 1983.

Miller and group of researchers revisited fuzz testing in 1995 (Miller, et al., 1995). This technical report detailed the testing methods used and results which were promising but still considered them as not good enough. One of the testing areas that proved the most promising results was networked applications which Miller et al. were not able to crash to the extent of offline programs. According to Takanen et al. (Takanen, Demott, & Miller, 2008) this sparked the Oulu University Secure Programming (OUSPG) group to develop testing suites for different networking protocols as detailed in "Vulnerability Analysis of Software through Syntax Testing" by Kaksonen et al. (Kaksonen, Laakso, & Takanen, 2000). Main idea of Syntax testing was to assure that the input syntax was correct and then fuzzed, henceforth guiding the fuzzer to fuzz areas of interest.

Guided approach uses different techniques to direct fuzzer to detect specific kind of vulnerabilities (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, pp.

1212-1213). This kind of fuzzing is an extension to syntax testing adding techniques like *dynamic taint analysis* (DTA) (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1207) and *control over execution flow* (Haller, Slowinska, Neugschwandtner, & Bos, 2013) of the program. Controlling execution flow needed a framework one of which most widely known is Valgrind by Nethercote and Seward (Nethercote & Seward, Electronic Notes in Theoretical Computer Science 89 No. 2, 2003); in which program is translated to x86-to-x86 just-in-time (JIT) compiled code, gaining control over the execution of the program. Valgrind is used in a fuzzer called Flayer (Drewry & Ormandy, 2007). Flayer used Valgrind´s dynamic binary instrumentation framework and memory error detection plug-in "Memcheck" to mark data and alter its flow. In dynamic data analysis data (whether its user inputted or generated through functions) is tagged with precision meta-data. This meta-data is then tracked through the program and removed when the original data is destroyed (Nethercote & Seward, ACM Sigplan notices vol 42, 2007). Therefore, in event of crash the input has a meta-data tagged to it which tells the user the propagation and origin of said input.

Guided fuzzing was improved with syntax aware fuzzers, for example "BuzzFuzz". Buzzfuzz was capable of processing files (instead of volatile inputs, like network data) which it dynamically tainted and therefore kept the syntax of the file intact. After initial tainting the fuzzer could improve on directing the fuzzing to upkeep the syntax while fuzzing revealing several bugs in Adobe Flash player and muPDF (a PDF viewer) (Ganesh, Leek, & Rinard, 2009). Syntax awareness was not new technique when BuzzFuzz was introduced as syntax aware fuzzers were first introduced in early 2000s by for example OUSPG PROTOS project, which had fuzzer suites for several networking protocols (Takanen, Demott, & Miller, 2008, p. 23). Adding DTA and control flow management extended the capabilities of fuzzers beyond protocol- and syntax testing to domain of file types.

While guided fuzzing was used in vulnerability searching in early 2000s, a different toolset was used in quality assurance. In quality assurance test-case generation was enhanced by incorporating automatic test generators. Thomas Ball analyzed a programs behavior by using symbolic execution to generate test cases that would traverse the behavior of "small inputs". In this case study, the input was array with a length of three to six (integers). Small inputs were chosen as exploring all paths of a program was deemed impossible (Ball, 2003). This problem in symbolic execution is called *path explosion problem*: larger programs have exponential number of paths (nested calls, loops, conditions) leading to scaling problems when traversing the whole program (Krishnamoorthy, Hsiao, & Lingappan, 2010). There are three ways to circumvent this problem: cut loop iterations, use small input space, or constrict the maximum path length (*bounded model checking*) (Ball, 2003).

Symbolic execution imbued with concrete execution to generate test inputs was introduced in application Concolic Unit Testing Engine for C (CUTE) (Sen, Marinov, & Agha, 2005). The author of CUTE stipulates that first known instance to bridge symbolic- and concrete execution were Larson and Austin in their presentation "High Coverage Detection of Input-Related Security Faults" (Larson & Austin , 2003), and CUTE was to apply this theory to a real-world test case generation for C-language.

CUTE was able to find problems in a single C-language function by either testing the function itself or constructing a framework to test the function (i.e. constructing main-function around tested function). The CUTE tests are generated and run per C-function meriting the name *concolic unit testing* (Sen, Marinov, & Agha, 2005). However, it is impossible to have an active environment present: any out of bound input (input that is not known when CUTE is launched) would be discarded, making the dynamic execution impossible. Dunbar et al. introduced dynamic execution for unit testing in their tool EXE which developed further to KLEE (Dunbar;Engler;& Cadar, 2008). KLEE used Low Level Virtual Machine (LLVM (Lattner & Adve, 2004) and its compiler (LLVM-GCC) to generate executable for KLEE to run tests on. This made the tool easily accessible as it compiled the whole program instead of testing single functions. While KLEE was making testing more approachable it did not try to solve the path explosion problem: it either chose a random path or chose a path that yielded most coverage to code (Dunbar;Engler;& Cadar, 2008).

Chipunov et al. proposed a solution in which the path explosion problem was solved by choosing the paths that seem important to the execution of binary and not explore paths that lead outside of the desired scope (*bounded model checking*). Path explosion problem was circumvented by using *Selective Symbolic Execution* (S2E). S2E used Quick Emulator (National Institute of Standards and Technology, 2018) (QEMU (Bellard, 2005)) virtual machine, KLEE-framework and LLVM tools as starting point, adding several thousand lines of code (KLOC). Developed environment runs on multiple operating systems (OS) including but not limited to MAC OS, Windows, and Linux. Chipunov et al. do not refer S2E as concolic, as it does not use concrete execution in exploring the binary path: all path constraints are assigned symbolically, and only when binary makes a call that is not defined by boundaries (i. e. call to OS or library), it is assigned a concrete value, in order to satisfy that path. S2E uses rules of execution consistency model proving state transitions from symbolic model to concrete execution and back. (Chipunov, Kuznetsov, & Candeae, 2011)

Godefroid et al. introduced fuzzers using *concolic execution* (symbolic and concrete) in *Directed Automated Random Testing* (DART) (Godefroid, Klarlund, & Sen, 2005). DART used symbolic execution to impose input vectors that satisfy the path constraint in order to find new paths for fuzzing. By combining symbolic and concrete executions deeper bugs can be found (Stephens, et al., 2016). Fuzzers can use concolic execution by reducing the scope of symbolic execution, prioritizing explored code paths, or controlling the amount of taint propagation (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, pp. 1207-1208).

First fuzzer to implement guided and concolic approach was Dowser (Haller, Slowinska, Neugschwandtner, & Bos, 2013, p. 50) in which the problems regarding path explosion were addressed by constricting the scope of symbolic execution to include only buffer boundary violations (also known as buffer over-

or underflows). Like the test generator KLEE Dowser also used LLVM as the environment but used S2E as an automatic path analyzer. Dowser was able to find several bugs in multiple Linux applications. (Haller, Slowinska, Neugschwandtner, & Bos, 2013, p. 59)

Concolic execution is a performance intensive task. Using only concolic execution induces a large overhead even with modern approaches: S2E using concrete execution induces approximately 6 times more the overhead than normal a virtual machine, and 78 times more overhead when using symbolic execution (Chipunov, Kuznetsov, & Candeae, 2011, p. 11). On the other hand, fuzzers produce only small amounts of overhead. Because they generate the input and run it on binary itself only inducing overhead when the input is generated, on monitoring the application, or in an event where a crash occurs (Nagy & Hicks, 2019). In Driller, the adopted approach is to use fuzzers to find new paths, and use concolic (or selective symbolic execution) to find paths when the fuzzer is stuck on finding a unique value (Stephens, et al., 2016, p. 3).

Driller is an all-purpose fuzzer unlike Dowser. Driller acknowledges that fuzzing a path is far faster than symbolic execution. The concolic execution engine of Driller is an open source software based on Mayhem (Cha, Averinos, Rebert, & Brumley, 2012) and S2E, and is called "angr". Driller works in four phases: first phase is to generate input test cases, secondly fuzz with said cases, thirdly use concolic execution to path exploration, and finally repeat the phases. Test cases are inputted first in order to guide the fuzzer towards wanted paths. Fuzzing starts with AFL mutated inputs, and concolic execution is invoked when a complex check or input is reached. Concolic execution starts by translating the binary to Valgrind "VEX" format in which symbolic constraints, states, and variables are defined. After transformation concolic engine analyzes the trodden path and starts solving the restraints therefore circumventing the path explosion problem. Repeat phase pushes the new path (and its inputs) down to the fuzzing engine and the process off Fuzz starts again. Driller was able to beat both symbolically executing Mayhem and pure fuzzer (AFL) instance in finding bugs in Cyber Grand: Mayhem found 16, AFL 68, and Driller 77 bugs. This proved that fuzzing and concolic execution work well together as a hybrid solution. (Stephens, et al., 2016)

Fuzzers have been used in wide variety of fields: test case generation, input testing, protocol and syntax testing and lately as an part of framework. It is clear that fuzzers are used because of their fast execution but they need more guidance to gain coverage on larger and complicated programs. AFL as a fuzzer is still used in modern day applications, like Driller, to its full extent.

2.1.2 General process of fuzzing

General process-schema has been depicted by Hongliang & al in Figure 3 (below) (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018). Fuzzers can be divided into two categories: *generational* and *mutational*. Generational fuzzer (also called grammar-based (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p.

1202)) is practically a ruleset which generates the input and observers its effect on the target program before generating a new input. Mutational fuzzer generates an input based on a valid source (either a network session, file, object etc.) by mutating it with different approaches (for example bit flips, adding or removing bytes). Most generation fuzzers are protocol, application or file-format specific whereas mutational fuzzers tend to be called generic- or general purpose fuzzers. (Takanen, Demott, & Miller, 2008, pp. 137-138)

All fuzzers require data input to start fuzzing. Creation of the inputted data is divided into four categories: test cases, cyclic data insertion, random data, and known attack vectors. Test case based input generation uses the same inputted data without mutation and can be considered more as a traditional automated (stress test) tool. Cyclic data insertion works mostly same way as test case based but it inputs data to test cases after a cycle is complete. Random data usually needs a starting point (from which to mutate) and the data insertion and mutation can be repeated. A fuzzer can also use known attacks, also called libraries, to test a given target program. (Takanen, Demott, & Miller, 2008, pp. 141-142)

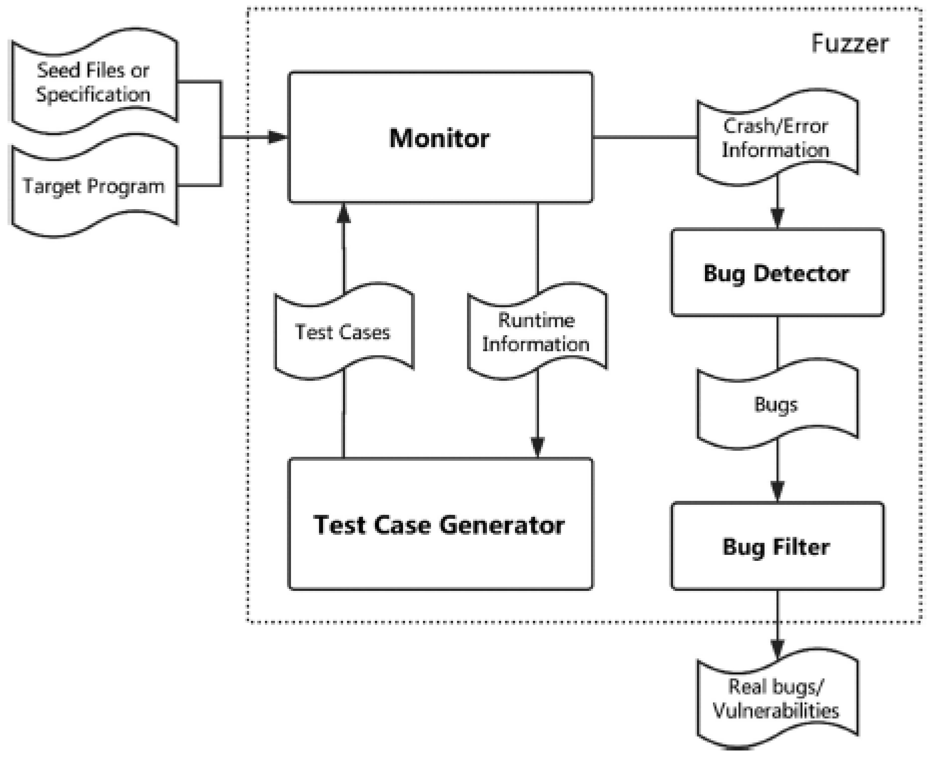


Figure 3 The general process of fuzzing according to Hongliang & al. (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018)

Seed files are a set of valid (or semi-valid) file inputs that are used in mutational fuzzers whereas specifications are inputted on generational (grammar) fuzzers (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202). Including randomness forces the target program to explore inputs that would not be possible according to program logic (for example, using different boundary values). A deterministic (“intelligent”) approach can explore the program more thoroughly but requires more time to set-up properly. (Takanen, Demott, & Miller, 2008, pp. 138-141)

Chosen target programs include for example: OS local program, Network interface, file ingesting applications, API´s or Web/server/client applications (Takanen, Demott, & Miller, 2008, pp. 162-164). Choosing the target program enables different kind of fuzzers with different kind toolsets, like concolic execution, protocol fuzzing, coverage guided fuzzing and such. On the other hand it is more likely that target programs information is limited in nature as it is most probable that the source code for target program is not available therefore limiting the choosing of different fuzzing techniques (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202). As input generation can be done in several ways it is most likely that choosing the target program is done by personal or some other interest.

The monitor component uses information from different sources to provide situational awareness to fuzzer. Amount of information sources used is specified by the target program (and its available information) and available runtime information (runtime errors, tainted data flows etc.). The monitor guides test case generator and receives and runs new test cases. Ideally, each test case would execute a different path but as this is not the case (if- and loop- conditions in code) the monitor must observe the *input gain* (IG) of a test case. This IG determines if a new path is taken or if the input is running in a loop, favoring new paths in coverage base fuzzers (Rawat, et al., 2017, p. 2). Finally, the monitor reports suspected crash or errors information (timeouts, system errors…) to a bug detector. (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202)

The *test case generator* (TCG) can be run with or without a monitor. When running without a monitor no runtime information is used and modified inputs run on the target program directly. TCG is either mutation- or grammar based (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202), and can be either *intelligent* or *non-intelligent*. Intelligent TCG possesses the understanding of a protocol (or file format), a target program interface, and necessary calculations to satisfy data integrity checks (for example, cryptographic functions) (Takanen, Demott, & Miller, 2008, pp. 142-144).

After an input performs a suspected crash, or error, the bug detector tries to validate said inputs crash inducing properties. Validation is sometimes called triaging (Woo, Cha, Goettlib, & Brumley, 2013). Bug detector collects and analyzes related information, for example collecting *stack traces* of crashes, or using debugger(s) to collect crash information. In Wikipedia the stack trace is defined as *"…is a report of the active stack frames at a certain point in time during the execution of a program"* (Wikipedia Commons, 2017). After information collection suspected bugs are de-duplicated (multiple inputs can induce the same bug, for example, in a loop or in if statement). Deduplication is usually done via stack hashes which can contain, for example: sequence of instruction pointers, debug symbols, names of function calls, line numbers of source code, object names etc. hashed with call stack functions (Molnar, Li, & Wagner, 2009, p. 8). After bugs are validated (ie.

confirmed that they induce an exception in target program, and are not duplicates of other bugs) they are passed to a bug filter. (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202)

The bug filter evaluates the exploitability of bugs. Exploitability is usually evaluated manually which requires low level debuggers and expert knowledge of target program (Takanen, Demott, & Miller, 2008, p. 31). The target program can be proven to be exploitable if the crash-inducing input causes an exploitable vulnerability. (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202)

2.1.3 Classifying fuzzers

As per Figure 3 (chapter 2.1.2 "General process of fuzzing"), a chosen target program, seed files, specifications or raw inputs conform to yield information to the fuzzer program. The fuzzer program executes, and this process yields anomalous program states that after classification are perceived as "bugs". This process leads to classifying fuzzers by their available information of target and runtime information. Commonly fuzzer programs are classified as white-, grey-, or black-box fuzzers. White box fuzzing has the most information, like protocol knowledge, compiled source code, known bad inputs, runtime code- and data-flow-coverage, or CPU and RAM utilization and shadowing. Greybox fuzzers rely on a binary that has been instrumented in some way, and black box fuzzers rely only on input generation and execution. (Takanen, Demott, & Miller, 2008, pp. 144-145) (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202)

Black-box testing refers to "unknown" codebase of which the program has been compiled (Takanen, Demott, & Miller, 2008, p. 144) and the input data is generally randomized from a well formed set of seed files. Black-box fuzzers are sometimes called "Black-box random testers" and are simple to use, making them popular in the software industry. Downsize to black box fuzzing is that no dynamic target information is modified or observed during runtime. Therefore, fuzzer does not excel in exploring paths that have low chance of triggering from random input (for example specific numbers and strings). (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202)

White-box testing includes testing methods that work on the source code itself, making the target applications code-base and combined inputs "known" variables. Process of white-box fuzzing relies on a reviewable codebase (Takanen, Demott, & Miller, 2008, p. 144), symbolic analysis and concolic execution to form path constraints for the coverage guidance. In theory, a white box fuzzer can traverse 100% of the program paths. In practice without an element of blind randomization (i.e. black box fuzzing) white-box fuzzing halts or slows down due to solving of path constraints during concolic execution (numerous paths, path explosion). (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1203)

Grey-box fuzzing is a step down from white-box: while the target itself is of unknown codebase, runtime information about code coverage can be lifted from the target program. A difference to black box fuzzing is that some codebase is known (for example, protocol, code- or script snippet) ascending the fuzzer above just random testing. (Takanen, Demott, & Miller, 2008, pp. 144-145). Two most common methods are code instrumentation to determine if a new path is found after *input mutation* or *taint analysis* which traces the dataflow to guide the mutation algorithm mutating specific areas of input. The methodology of white- and grey-box fuzzing is quite similar (both make use of runtime information to guide the fuzzer for more coverage) but the main difference is that a grey-box fuzzer does not the utilize source code of target program, unlike the white-box fuzzer. (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1203)

2.1.4 Modern coverage guided fuzzer (AFL)

AFL can be classified as a unintelligent general purpose fuzzer according to Takanen et al. because it can be used to test multiple interfaces, but does not have deep understanding of said target interface (Takanen, Demott, & Miller, 2008, p. 149). Hongliang et al. classify AFL as *a mutation-based coverage-guided fuzzer* (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202) which is closer to the authors own view of a *instrumentation-guided genetic fuzzer* (Zalewski, American Fuzzy Lop (2.52b), 2016). AFL can be dissected according to Figure 3 (chapter 2.1.3) to its components and their functions can be explored in the same context.

As a mutational fuzzer, seed files must be presented to the fuzzer along with the target program invoking command which also specifies where the input is inserted (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202). Seed files are very application specific but some common rules apply: selecting (valid/semi valid) files with different coverage are better than randomly generated, and reducing a set of files is more efficient and transferable than a full set of files (Rebert, et al., 2014). Theses seed files can be collected manually or via web crawler, randomly generated, or using data-driven approaches like Skyfire (Wang, Chen, Wei, & Liu, 2017) and AFLSmart (Pham, Böhme, Santosa, Roychoudhury, & Câciulescu, 2018). The quality of seed a input has great impact on performance, as proven with the case of AFLSmart versus AFLFast (Pham, Böhme, Santosa, Roychoudhury, & Câciulescu, 2018): AFLSmart (Seed input generation done with Peach-pit (Deja Vu Security, 2014)) found twice the amount of zero-day bugs than AFL with AFLFAST-extension (Böhme, Pham, & Roychoudhury, 2017) on the same target programs.

AFL is a grey-box fuzzer utilizing binary instrumentation (either done with AFL´s own C-compilers (AFL GCC and -CXX) when the source code is available, or QEMU user space emulation for *on-the-fly instrumentation* to monitor coverage on an application. AFL can also run in a black-box mode (only mutations, no runtime information), however this is not recommended. As per normal Greybox fuzzer the monitored coverage information is relayed to test-case generator to assist coverage-based fuzzing. (Zalewski, AFL Readme, 2017)

AFL test generator receives an input and tries to determine if input has gained coverage. Inputs performance cost is calculated and low performance cost coverage gaining inputs are favored for mutation. Mutation is done through bit flips, arithmetic operations, and interesting values, in set places (dictionary) or randomly (*Havoc*) by either splicing into or adding to the input. The Inputs are then set to an input queue to be executed by the target application. (Zalewski, AFL technical details, 2017) (Google LLC, 2019, pp. 5131-6659)

Bug detection in AFL is done after *fork-server* (Wikipedia Common, 2019) is up and ready to process the queue (in the "monitor" part). Queues input is then executed and checked if it crashes the target program (*execv* returns -1 (Linux Foundation, 2019)). AFL does not use stack trace rather it hashes the error code with its coverage information (stored in a *bit map*) and a constant to determine if the bug is a duplicate (Google LLC, 2019, pp. 2100-4554). This method is less accurate than the stack hash: if an identical crashing input crashes on multiple locations (for example, case structures, loops, mangled "if" sentences) of said bitmap it is considered unique and therefore inflates the crash count. As the deduplication is dependent on the amount of information gained from the target and information used to deduplicate it has been shown that AFL's own deduplication logic bloats the results when compared to stack hashes (Klees, Ruef, Cooper, Wei, & Hicks, 2018, p. 2124). AFL does not have a bug filter. Therefore any crash that is reported by AFL must be validated as vulnerable by either manually or using tools. (Zalewski, AFL technical details, 2017)

2.1.5 Fuzzing target

As mentioned in Chapter 2.1.2 General process of fuzzing, the fuzzing target can be either in binary- or source code form (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018) and targets cover from real-world applications to synthetic benchmarks (Klees, Ruef, Cooper, Wei, & Hicks, 2018, p. 2126). These targets are usually chosen from modular open-source projects and standard benchmarks because they are readily available and relatively simple to test (Liang, Wang, Chen, Jiang, & Zhang, 2018, p. 562). The examples of these targets that have been used in academia to compare fuzzers are listed below on Table 1 (Klees, Ruef, Cooper, Wei, & Hicks, 2018, p. 2127) (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202). Additionally, ten interesting targets and their listed *CVE* identification numbers (Common Vulnerabilities and Exposures (MITRE, 2020)) are compiled to Table 2 (far below) (Zalewski, American Fuzzy Lop (2.52b), 2016). Table 2 is in non-exhaustive of the accomplishments of AFL.

Table 1 Examples of fuzzing targets from academia

Real-world applications	**Synthetic Benchmarks**
Binutils 2.2.6: nm, objdump, cxxfilt	Cyber Grand Challenge 2016
gif2png	LAVA -M
FFmpeg	
png2swf	

Table 2 AFL "bug-o-rama" targets.

Target	CVE's
Mozilla Firefox	CVE-2014-1564 CVE-2014-1580 CVE-2014-8637
Internet Explorer	CVE-2014-6355 CVE-2015-0061 CVE-2015-0076 CVE-2015-0080
OpenSSL	CVE-2015-0291 CVE-2015-1788 CVE-2015-1789 CVE-2015-3193 CVE-2016-2108
tcpdump	CVE-2014-8767 CVE-2014-8768 CVE-2014-8769 CVE-2015-3138 CVE-2016-7993
curl	CVE-2015-3144 CVE-2015-3145 CVE-2017-7407
PHP	CVE-2015-0232 CVE-2017-5340
BIND	CVE-2015-5477 CVE-2015-5722 CVE-2015-5986
Apache httpd	CVE-2017-7668
irssi	CVE-2017-5193 CVE-2017-5196 CVE-2017-10965 CVE-2017-10966
clamav	CVE-2014-9328 CVE-2015-1463 CVE-2015-2170

Choosing a target for assessing performance of a fuzzer differs from using fuzzing as a bug discovery tool as the developers or vulnerability researchers might use. The performance is usually assessed (in fuzzer research) against a ground truth. Therefore, the target is better to choose from targets that yield consistent feedback to both the fuzzer and the experiment. This requirement limits the target to either a synthetic benchmark or to an application with known bugs, as ground truth might be hard to establish with real targets. (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2124,2127-2129)

It is most likely, that the target has more than one bug. Both real-world applications and synthetic benchmarks have *shallow* and *hidden* (deep) bugs (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1204). Shallow bugs crash the program most frequently, while residing in the most frequent paths used by the program (Liang, Wang, Chen, Jiang, & Zhang, 2018, p. 564), for example, a divide by zero operation in a non-conditional path (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1204). Bugs that are deep in the complex conditional branches of a program are harder to find (hence the name hidden) (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1204). These bugs also elude the fuzzer as they might not crash expectedly (for example, a logging function takes over and logs the crash before a bug detector can detect the crash (Liang, Wang, Chen, Jiang, & Zhang, 2018, pp. 564-565)), or the crashing path requires a sequence of unique (*magic*) bytes or conditions to be executed (Stephens, et al., 2016, pp. 4-6). As there is no standard way to detect if a bug is either shallow or deep in the program code the evaluation of depth is done through code coverage (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1204). AFL measures the code coverage by instrumenting the compiled application through its own compilers and therefore the target application must use Linux as the OS. Furthermore, the source, or pre-instrumented code must be available. Besides a swathe of Linux utilizes that can be fuzzed there are two prominent synthetic benchmark suites: *Large-scale Vulnerability addition* (LAVA) and *Cyber Grand Challenge 2016* (CGC) (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2124,2127).

LAVA is a Linux based framework, written in C-language, consisting of three parts: Proof of concept LAVA-1, benchmark LAVA-M and synthetic bug injection toolchain LAVA. LAVA-1 is a Linux utility file which has been injected with 69 buffer overflow bugs of which 20% was found by the tested fuzzer within a five-hour period. LAVA 1 proved that buffer overflow type of synthetic bugs can be inserted into a program. The second framework called LAVA-M consists of four linux coreutils 8.24 programs injected with various amounts of bugs: base64 (44), md5sum(57), uniq(28) and who(2136). For this framework, the fuzzer found seven bugs from base64, seven from uniq, two from md5sum and none from who in a five hour period, although a bug was found from who at sixth hour. The last framework LAVA allows bug injection to arbitrary C-language programs for creating your own bug corpora. (Dolan-Gavitt, et al., 2016)

Cyber Grand Challenge (CGC) was a *DARPA* (Defense Advanced Research Project Agency) led competition that focused on vulnerability discovery and patching automated solutions that span multiple architectures over an network interface. The goal of the challenge was to be able to stretch the vulnerability discovery and patching concept to machine speeds while motivating the 100 teams with competitive prices of up to 2 million dollars. In 2016 the challenge was won by Carnegie Mellon University with their *Mayhem AI* (Not to be mixed with *MAYHEM* concolic executor) (Coldewey, 2016). The corpus is available at Github (Github, 2018). (Fraze, Dustin, 2020)

While CGC would provide an interesting example for both virtualization and AFL this study will use LAVA-M's who utility as the target program as it didn't provide a bug within normal time limits of experiment rather in 6 hours. Said result can be compared with this study's results to provide insight on if the LAVA-M used fuzzer was comparable to AFL or was it AFL itself.

2.2 Performance theory

Considering the motivation of this study to do efficient fuzzing in offline environments (see Chapter 1.1) the measurement of performance is crucial. Performance can be measured through many different metrics which are overviewed in Chapter 2.2.1 alongside this study definition of performance. From metrics we can derive the scalability of an instance, which in short is the presumption of how much power is gained when workers are added in parallel to work on same task (McCool, Robinson, & Reindeer, 2012, pp. 55-56). As we know what needs to be done for scalability metrics, we can stipulate if AFL can provide these metrics and how they could be harvested.

2.2.1 Performance

Performance is completing a measurable task (also called *work*) in a limited timeframe. Performance can be identified as lacking (when not being able to complete a said task in a timeframe), abundant (when increase in performance does not yield more results in timeframe), or anything in between. Need for performance arises from either being able to reduce cost of computing, or being able to compute tasks faster: reducing costs is gained when more work is done with same hardware (or power budget), and increasing total amount of work usually makes a system produce results faster (when task is not insurmountable*). Task parallelization* is considered a performance-enhancing factor by either (McCool, Robinson, & Reindeer, 2012, pp. 54-58):

1. Reducing (or hiding) latency,
2. Increasing throughput (rate of completing tasks),
3. Using less power to compute results.

Latency is defined as time to complete a task, and therefore (in performance) lower latency is usually better. *Response time* is related to latency: it is used to measure the time between systems, to check if a process (between systems) is done in a set time. Latency is closely related to throughput: increasing, decreasing or hiding latency affects a systems throughput. (McCool, Robinson, & Reindeer, 2012, pp. 55-56)

Throughput is defined as series of tasks completed in measured time; therefore, it is the rate of work and time: work per unit of time. On the contrary of latency, larger throughput is better. Closely related term to throughput is *bandwidth*, which refers to work per frequency rate used in many memory or communication applications.

Power consumption is "*the sum of dynamic power consumption and static power consumption*" (McCool, Robinson, & Reindeer, 2012, p. 57). The power consumption can be managed by adding more resources, on the underlying condition that the task itself is parallelizable. Power consumption can also be managed by "*racing to sleep*" in which task is done as fast as possible, in order to save energy by sleeping. (McCool, Robinson, & Reindeer, 2012, pp. 57-58)

Parallelism is usually induced into system by either pipelining or latency hiding (queueing). Pipelining increases latency and throughput by overlapping different series of tasks. Overlapping is controlled by synchronizing said tasks in parallel, which by definition increases latency (overhead of single task increases). Latency hiding is not hiding the latency, rather a process switching mechanism, in which a process has several options on to complete (can also be called "process queue"). When a process has to wait, it changes to different process, therefore "hiding" the high latency (or response time) of previous process. (McCool, Robinson, & Reindeer, 2012, pp. 55-56)

2.2.2 Performance metrics

When measuring a performance gain two metrics are used: *speedup* and *efficiency*. Speedup is the comparison of single worker latency (i.e. time taken to solve a computational problem) to multiple workers combined latency (on the same problem). Efficiency is measured by dividing speedup by the number of workers. Equations for speedup and efficiency are presented below in Equation 1 where T_1 is latency of one worker, T_P is the latency of P workers, S_P is speedup, and P is the number of workers. Speedup can be categorized to relative-, absolute-, *linear-*, or *sub-*, or *superlinear-speedup*. (McCool, Robinson, & Reindeer, 2012, pp. 56-57)

$$speedup = S_P = \frac{T_1}{T_P}$$
$$efficiency = \frac{S_P}{P} = \frac{T_1}{PT_P}$$

Equation 1 speedup and efficiency equations (McCool, Robinson, & Reindeer, 2012, p. 56)

Algorithm that runs P times faster on P workers can be considered linear, whereas algorithm that runs X times P faster on P workers can be considered superlinear or sublinear depending if X is greater than one (superlinear) or less than one but greater than zero (sublinear). Both linear and superlinear speedups are rare because of coordination of tasks (and synchronizing them) contributes overhead when using multiple workers. Superlinear speedup is usually observed if one worker is resource limited (for example, multiple workers use cache better) or its parallel algorithm is somehow more efficient than serialized one (for example, tree search algorithms). (McCool, Robinson, & Reindeer, 2012, pp. 56-57)

When a parallel algorithm worker is used as a single worker it is *serialized* and its speedup is reported as relative speed up compared to *non-serialized* worker. If a non-parallel (i.e. a worker which algorithms works better serialized than in parallel) worker is used is used in parallel to determine speedup in multi-worker environment the metric is called absolute speedup. Both absolute and relative speedups are algorithm agnostic, as long as the problem (i.e. target) is the same in both calculations. (McCool, Robinson, & Reindeer, 2012, pp. 56-57)

When ether the speedup is absolute or relative, in quantity sublinear speedup is the most prevalent outside hardware restricted parallel workers (which as stated can show linear or superlinear behavior). Speedup is the metric that provides data on how scalable a worker is.

2.2.3 Scalability

Scalability is the rate of speedup to worker amount (*efficiency*) in a limited problem. Limits to scalability are depicted in Amdahl´s law and Gustafsson-Barsis´ law, having strong scalability and weak scalability respectively. Scalability can also be estimated by using worker-span model, which instead of latency uses critical path and span of the algorithm as basis for determining speedup. (McCool, Robinson, & Reindeer, 2012, pp. 57-64)

Both Amdahl and Gustafson-Barsis state that programs work is divided into two partitions: parallelizable and non-parallelized (serialized). In Amdahl's law parallelized work is a set pool of work which is evenly distributed to multiple workers and therefore the efficiency of target a program is limited by the serialized portion of algorithm. Whereas Gustafson-Barsis' observations indicate that parallel workers allow for more work done in general therefore the speedup is not limited by the serialized work but by the number of parallelized workers. This difference in observation is demonstrated below on Figure 4. Both laws e equations are presented in Equation 2, far below. (McCool, Robinson, & Reindeer, 2012, pp. 58-62)

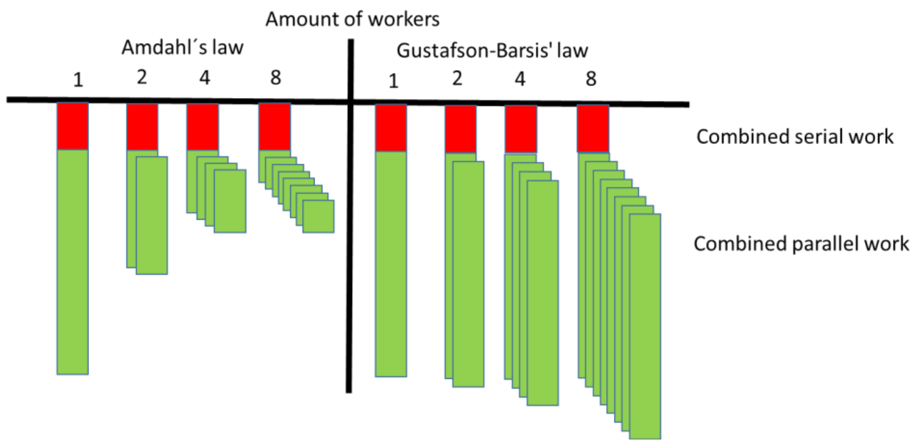


Figure 4 Amdahl and Gustafson-Barsis´ law view on combined work (McCool, Robinson, & Reindeer, 2012, pp. 59,62)

$$S_P \leq \frac{W_{SER} + W_{PAR}}{W_{SER} + {}^{W_{PAR}}/_{P}}$$

$$S_{PS} = W_{SER} + P \times W_{PAR}$$

Equation 2 Amdahl´s and Gustafson-Barsis' law mathematical presentation (McCool, Robinson, & Reindeer, 2012, p. 59) (Gustafson & Barsis, 1988)

Amdahl's law cannot exceed linear speedup but can exhibit linear speedup in cases where no serialized work is present (McCool, Robinson, & Reindeer, 2012, p. 59). This is demonstrated through Equation 2 (above) top equation which shows that even a fraction of serialized work drops the total speedup below linear speedup when multiple workers are used. With Gustafsson-Barsis's law superlinear speedup is attainable if serialized part of the work is less than parallelized part.

Every real-world system has a scalability limit, which manifests by either hardware or software restriction. These restrictions cause imperfect parallelization. In this kind of system, worker-span model can be used to estimate the upper- and lower bounds of scalability. Worker-span model relies that the program has greedy scheduling (it does not allow the worker to stand idle) and its *span* (also called *step complexity* or *critical path*) can be determined. (McCool, Robinson, & Reindeer, 2012, pp. 62-65)

2.2.4 Scalability and AFL

In this study AFL´s scalability is evaluated through two performance theories: Amdahl´s- and Gustafson-Barsis´s law. Worker-span model is not used as it would require constructing the theory on how critical path is counted in fuzzing applications where needed results are crashes which itself is a great area of study for future research.

In current era of *x86* (and *x64*) based processor architecture multiple cores (and core-threads) are used to enhance the computing power and efficiency of computer system. In architecture this is achieved by either using cores as reserve computing power (being more energy efficient) (Dong & Hsien-Hsin , 2008) or using symmetric or asymmetric processor structure (Hill & Marty, 2008). Most notable asymmetric structure being System on a chip (SOC) type of structure (Paul & Meyer, 2006). Both Amdahl and Gustafson-Barsis's law struggle in these environments as they were conceived in an era where computer systems were more or less heterogenic due to their construction and physical size (Paul & Meyer, 2006, p. 104). This leads to following restrictions:

1. Symmetric processors are used (i.e.. no helper threads or *hyper threading*) (Hill & Marty, 2008, p. 37),
2. Architecture is not allowed to throttle cores (Dong & Hsien-Hsin , 2008, p. 30),

3. Using both laws to asses results as problem size is hard to quantify: Amdahl's law for small problem sizes and Gustafson-Barsis law for large problem sizes. (Hill & Marty, 2008, p. 37).

First and second restrictions are handled through the software design of AFL itself and by using a symmetrical processor for experiment. AFL requests a CPU (thread) for itself on launch if some delay on multiple instance launch is introduced: without a delay (i.e., using tools like afl-launch) the AFL will not bind correctly to a CPU (thread). This leads to a situation where many AFL processes are bound to single thread leading to over congestion of the thread and performance issues (Gamozo Labs, 2018). Throttling of the cores is checked by AFL before startup and AFL will not launch if it finds that the CPU speed governor is not set on performance mode which does not allow the CPU to optimize power consumption through throttling core speed (Google LLC, 2019, pp. 7327-7380).

Third restriction is vague on purpose: using Amdahl's and Gustafson-Barsis's law for performance testing a fuzzing application is not a straightforward case with any fuzzer. AFL is built to handle parallelism through several features: queue to communicate with other instances (so a worker never runs out of work), deterministic checking to advance the coverage (aka. "Master instance") and non-deterministic behavior to fuzz new coverage (aka. "Slave instance") (Google LLC, 2017) and propagation of coverage information between deterministic and non-deterministic instances (Xu, Kashyap, Min, & Kim, 2017, pp. 3-5).

Known bottlenecks for AFL performance are its utilization of fork system call, file system operations (creating, opening and scanning files) (Xu, Kashyap, Min, & Kim, 2017) and requesting CPU cores from OS (Gamozo Labs, 2018). Xu et al. have mapped the performance of AFL from one to 120 cores, which in this case is considered data center-level of computing power. Metric in this study was conceived by mapping the execution speed as a function of time over core count. Results are underwhelming as execution speed starts to drop after 30 cores and completely collapses after 120 cores (Xu, Kashyap, Min, & Kim, 2017, p. 4).

Xu et al. research does not yield any information of what is the speedup or scalability from one core to 120 cores. Nor does it conform to Amdahl or Gustafson-Barsis law definition of scalability as a function of speedup. Research only uses this quick method of measuring scalability as function of executions (of target program with mutated input) per second as a basis of designing new operative primitives for yet another AFL derivative. Both laws state that speedup (main component of scalability) is a function of serial and parallel work between workers.

Goal of fuzzing is to find an input that crashes target program. If we presume that the proof of work in fuzzing is a crash, not successful execution (executions per second) of said input, we can measure the time it takes for workers (from one to infinity) to produce a crash. This crash can then be deduplicated through bug filter, compared to larger instances exhibiting the same crash yielding speedup between the instances, conforming to Amdahl or Gustafson-Barsis's law. These results could contradict using execution speed as a metric for scalability.

2.3 Virtualization

Virtualization is heralded as an technique that allows multiple users to use same hardware resource through sharing it (Nanda, Chiueh, & Stony, 2005, p. 2). Therefore, using virtualization could be effective tool to contain and isolate fuzzing instances. In this chapter virtualization is defined for this study and its virtualization levels are outlined in order to make an decision for virtualization in the experimental setup.

2.3.1 Defining virtualization

> "Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others" (Nanda, Chiueh, & Stony, 2005, p. 2) (Singh, 2004)

Virtualization technology was developed from *time-shared mainframes* where users connected (with terminals) to large mainframes having a set amount of memory and processor capability in their disposal for set amount of time. Time shared computer-systems shared lot of the same features that are now common *virtual machines* (VMs). (Nanda, Chiueh, & Stony, 2005)

In both time-sharing and virtualization environment the features are tooled towards isolating the underlying system from both user faults and foul play. In time sharing systems, this is done through operating system (and its hardware specific components (Bell, 1967, pp. 1-7)) whereas in virtualized system isolation is done through instruction set-, hardware-, operating system-, programming language, or library level of abstraction. (Nanda, Chiueh, & Stony, 2005, pp. 6-23)

Virtualizing application and hardware interfaces imply that the interface itself is not real, rather a faux interface that mimics the capabilities and restrictions of a real interface. Originally, virtualization was used to execute applications (or different architectures) on computer without the fear of crashing the bare-metal hardware but it has evolved to include multitasking and other partitioning (*isolation*) or aggregating (*converging*) applications. (Nanda, Chiueh, & Stony, 2005, p. 2)

2.3.2 Virtualization levels

Most commonly virtualization is considered to isolate systems from others simultaneously having executive rights to users in their own isolated segments. Practical scenarios include (but are not limited to) server-, and application consolidation, sandboxing, multiple execution environments or simultaneous operating systems, virtual hardware, debugging, software migration, appliances, and testing and quality assurance (QA) (Nanda, Chiueh, & Stony, 2005, pp. 2-3). Virtualization can be divide by its execution environment (Kovari & Dukan , 2012) or by abstraction level (Nanda, Chiueh, & Stony, 2005, pp. 6-23) as presented in Table 3 below from low to high level applications with example technology frameworks.

Table 3 Virtualization by execution environment and abstraction level

Abstraction level	**Execution environment**	**Examples**
Instruction set	Processor emulation	Bochs, Crusoe, QEMU
Hardware layer	Full -, partial -, hardware assisted-, and multi-server (cluster) virtualization	VMWare, KVM, and Xen
Operating system	Para virtualization, operating system-level virtualization	(Free BSD) Jail, Solaris zones/containers, OpenVZ
Programming language		Java Virtual Machine
Library		Wine, WABI

Instruction set level of virtualization implements emulation of different (from residing instruction set) architecture instructions in software. Usually this means translating guest machines instructions to native machine instructions and executing them. This is different from hardware layer (and subsequent layers) virtualization as whole system (including for example, input / output (IO) devices, ROM chips, rebooting) must be emulated from another instruction language to other whereas in hardware emulation same instruction set can be used. Instruction level of virtualization has big performance penalty, but provides multiplatform capability if instruction set does not change. (Nanda, Chiueh, & Stony, 2005, p. 6)

Hardware Abstraction Layer (HAL) virtualization provides the full, partial or consolidated performance of a system to one or more virtual machines (VM) by *virtual machine monitoring* layer (VMM). VMM is usually layered between the hardware and operating system in order to trap multiple incoming *privileged instructions* from other VMs to execute them on underlying hardware (Nanda, Chiueh, & Stony, 2005, p. 8). Therefore, the execution environment can be partially or fully virtualized compromising of multiple computers (*cluster*) still sharing the same level of abstraction. Prominent processors also provide hardware assisted virtualization (Kovari & Dukan , 2012, p. 336). Common way of using HAL in computer environment is to use *kernel-based virtual machine* (KVM), which is a Linux native module capable of spawning virtual machines as regular Linux processes in either guest, user, or kernel mode (depending on needed privileges). User mode is the default application, and is elevated to kernel mode if user needs services from kernel. (Kovari & Dukan , 2012, p. 336)

Operating system (OS) level virtualization creates a VM that has the same operating system as the host machine but isolates the guest VM from said OS by controlling the operating environment consisting of: user-level libraries, other applications, file system and data structures, and other environmental settings.

This creates a system that has multiple execution environments that are isolated but running on the host kernel (Kovari & Dukan , 2012, p. 336).Having multiple execution environments running the same Kernel mitigates the HAL virtualizations need for duplicate services where each virtual machine needs its own operating environment to be virtualized per machine which creates a performance overhead. Isolation of guest VM is done through partitioning and multiplexing the operating environment to gain fairly isolated operating environment from the host machine. OS level virtualization is commonly referred as *containerization* but is not limited only to this technology (Kovari & Dukan , 2012, p. 336). (Nanda, Chiueh, & Stony, 2005, p. 17)

Programming level virtualization is done in application level, where the programming language supports self-defined instructions that use the host hardware by either special I/O instructions or manipulation of memory. As an example, Java Virtual Machines (JVM) use memory manipulation and self-defined instructions (java byte codes) to execute a program in a state of isolation from host operating environment by interpreting java instructions through just-in-time Compiler (JIT). (Nanda, Chiueh, & Stony, 2005, pp. 21-22)

Virtualization on library level is done to hide the details of *application programming interface* (API) or *application binary interface* (ABI) from the user, in order to simplify a process. Library level of virtualization works above operating system layer to produce a different environment depending on the interfaces used to make a different API / ABI available. This kind of virtualization is sometimes called *ABI/API emulation*. (Nanda, Chiueh, & Stony, 2005, p. 23)

2.3.3 Fuzzing and virtualization

Fuzzing is a process of finding vulnerabilities in software. Therefore, depending on target isolating the process from host machine can be of importance. Fuzzing components have been virtualized from the instruction set level to OS-level of virtualization. The main question in virtualizing component (or target) is how the virtualization is used and to what problem is it trying to solve?

Instruction set level of virtualization is commonly used to emulate another processors instruction set over other. In fuzzing, the target itself usually is an application, but there has been some non-academic work fuzzing the processor instruction set (namely VIA C3 x86 instruction set (Domas, 2018)) nevertheless, fuzzing is commonly done on the same architecture as the target. Instruction set emulation (and virtualization) is still a tool that can be used in fuzzing as a feedback / fuzzing monitor to black-box fuzzing. For example: AFL implements QEMU in user mode to instrument black-box binaries during runtime, with a large performance hit due to emulation (Zalewski, AFL QEMU Readme, 2017). This induced on average 612 % overhead which is very big compared to other black-box techniques, for example: static binary rewriting (AFL-Dyninst) overhead is 518% and Dyninst based binary rewriting incorporated with interest oracles (UnTracer) averages at 0,3% overhead (Nagy & Hicks, 2019, pp. 1-2). Purpose of this research is to evaluate AFL´s performance in with grey-box methodology as we instrument the target binary with AFL´s own compiler. Therefore, this kind of virtualization and approach to fuzzing is not applicable because it has large overhead.

Full and partial virtualization would provide isolation and configuration management to virtual machine where the fuzzer and target lie. In this kind of scenario, the target would be isolated from other instances and malleable between different test cases. Furthermore we can be explicitly sure, that AFL has bound itself to an CPU in the virtual machine whereas in native environment it might bind itself to a thread (and therefore with more than one instance would be susceptible to queue the fork privileged instruction with the shared core) . Hardware assisted cluster (for example ProxMox (Kovari & Dukan , 2012, p. 338)) with kernel virtual machine (KVM) would solve some problems regarding AFL: binding of CPU, control of the OS environment, and forking. AFL´s fork mechanic is essentially a privileged instruction (Xu, Kashyap, Min, & Kim, 2017, p. 4) as it creates a new process. Therefore, having a Virtual Machine Monitor (*hypervisor*) greedily handing out the privileged instructions from the kernel level virtual machines to hardware level could conveniently solve forking problem.

Operating system level virtualization would furthermore lessen the overall need for virtual machine environment duplication by providing a single operating system environment with an isolated fuzzing target and fuzzer. This could be done via containerizing the fuzzer and its target, for example with AFL-docker containers (Carlton, 2017). This would not solve any performance bottlenecks as the docker container would still run on the same kernel as other parallel fuzzers and therefore would not solve any other problem than isolating the target and fuzzer from operating environment. The performance overhead should be less than in full, or partial virtualization but it still would incur a performance penalty compared to native execution. There are too many open questions in container fuzzing and combined with lack of available software it is not applicable to this research.

This research focuses itself on Linux systems, as both the target and fuzzer are Linux operable. As we are limiting the research to fuzzing target that is within the same operating environment as the fuzzer no programming level or virtualization level isolation is necessary. This forces these two virtualization levels out of scope for this research.

3 Methodology

As the overview of related fields are done, we must define the methodology that is used in this study. As per chapter 1.3 the chosen methodology of multiple-case study is overviewed more deeply in Chapter 3.1 and its data collection procedures in Chapter 3.2. Guidelines on how data from this study is processed outlines Chapter 3.3. Finally, this chapter answers the questions on validity and reliability in Chapters 3.4 and 3.5.

3.1 Overview

Research design is overviewed by evaluating multiple-case study methodologies compatibility with fuzzing. After this the multiple-case structure itself is constructed in Chapter 3.1.2. Finally, in order to be able to use said methodology and procedures they are manifested through constructing the experimental setup in Chapter 3.1.3

3.1.1 Research design

Case study relies on a clear chain of explaining a set of decisions, the reasoning behind them, implementation and results of said set of decisions. These decisions can manifest themselves on common cases for example, such as organizations, processes and events. What all of these have in common are that they investigate contemporary phenomena of which boundaries and context are ephemeral and investigation relies on multiple data points from multiples sources. The data points (hopefully) converge to results that support or disclaim previous theories about same kind of phenomena. (Yin, 2014, pp. 15-17)

As previously mentioned, data points can be plentiful and therefore some of them are bound to contradict the convergence to a theory. Therefore, a case study needs a clear research design in order to prove a logical path from data to results, while providing the causality between converging data points and disproving the non-converging data. Research design is a plan that is of a *logical model of proof that allows the researcher to draw inferences concerning causal relations among the variables under investigation* (Yin, 2014, p. 28). Case research design consist of five parts (Yin, 2014, p. 29):

1. a case study's questions
2. its propositions
3. its unit(s) of analysis
4. the logic of linking the data to the propositions
5. the criteria for interpreting findings

As discussed in Chapter 1.2.1 the main research questions revolve around AFL and its measurement of scalability in multiple instances configured in either master (deterministic) or slave (non-deterministic) configuration on both native and virtualized execution environments. Research questions were formed from own interest in fuzzing and consolidated during the literacy review of fuzzing, fuzzing instances and virtualization of said instances. As virtualization has become sort of a norm in modern computing environments it is imperative to know what kind of configuration and how many cores (virtual or native) are best for fuzzing application. Moreover, the term scalability was poorly understood in literature, mainly being mirrored through the execution speed of fuzzer given a non-trivial target using ever-increasing core count (Xu, Kashyap, Min, & Kim, 2017). This kind of testing is not even recommended in AFL documentation (Google LLC, 2017) and therefore resembles more about using fork to eat a soup rather than questioning how the soup must be eaten. Theory of performance and its scalability models are used to study the speedup of different instances. These theories implicate that if the speedup is linear but not superlinear then model of Amdahl's law implicates weak scalability of AFL whereas in the case of linear and superlinear speedup the model of Gustafsson-Barsis´s law implicates strong scalability. Using these models as the basis of this study gives the study context that is not previously explored on fuzzer research.

Proposition of this study leans heavily towards Klees & al article on how to assess fuzz testing (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2131-2132) and on Arcuri & Briand on using statistical test to assess randomized algorithms in computing (Arcuri & Briand, 2011). Klees & al rationalize several metrics with their pros and cons and of these metrics two metrics are used in this study: coverage reported by the fuzzer and (unique) bug count using stack hashes both addressed in Chapters 3.2.3 and 3.2.2 respectively. Execution speed is used as a competing metric as according to Xu et al. linear speedup should be apparent until 15-30 used cores (Xu, Kashyap, Min, & Kim, 2017). Proposition of the study is also further restricted by choosing a coverage guided fuzzer in Linux OS, as it also restricts the scope to x86-64 architecture computers. Therefore, proposition directs this study to examine the said metrics and their interdependencies in both scalability models in order to decide which one would model the scalability of AFL most precisely.

In this study the unit of analysis is defined through Hongliang & al general process of fuzzing (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018, p. 1202) which is presented in Chapter 2.1.2 and illustrated in Figure 3. Same general process is dissected regarding AFL in Chapter 2.1.4. In order to be able to produce repeatable results, a non-moving target for AFL must be represented. This target is LAVA test suites who program (Dolan-Gavitt, et al., 2016) which also provides the seed file for AFL eliminating the need for seed file gathering and simplifying the process of fuzzing. Figure 5 far below shows how this unit of analysis can produce data from applying AFL to general process of fuzzing (Hongliang, Xiaoxiao, Xiaodong, Wuweu, & Jian, 2018) and what data in which process phase can be extracted.

According to Yin the units of analysis must be bound in order to determine the scope of data collection (Yin, 2014, pp. 33-34). In this study binding is done through the case structure consisting of different cases having inner configuration changes and interdependencies to other cases in the contextual execution environment (virtual or native environment). As an example, AFL can be configured in a four-core instance in multiple ways: one master and three slaves, two masters and two slaves and finally three master and one slave. In this example, the metrics would be the same, but their values and statistical differences would answer if configuration change (number of masters and slaves) would change the performance of this four-core configuration. Meanwhile assessing the speedup of four workers in the best and worst configurations relative to one worker case yields information on how scalable four worker instance is compared to one worker instance. If virtualization hinders performance it will show by comparing units to each other, as they essentially are run in the same way in both native and virtual environment and therefore can be compared. Restrictions to a single unit of analysis (i.e. worker instance) are furthermore explored in Chapter 3.1.3 while the metrics on which the data is analyzed are detailed in Chapters 3.2.1- through 3.2.4.

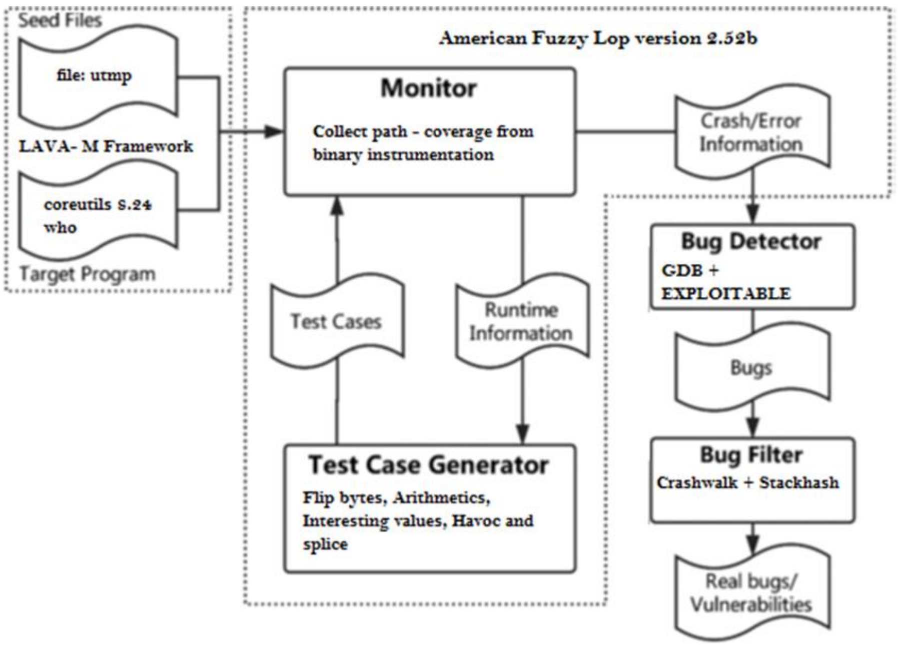


Figure 5 Unit of analysis in this study

Case structure is built on virtualization and native execution cases having a different context because it is most probable that virtualization occurs a performance overhead that affects speedup measured and therefore affects the scalability measured from units of analysis. Therefore a multiple case design (type 4 design as mandated by Yin (Yin, 2014, p. 50)) is used having two contexts: virtual- and native execution. As the cases differ only by variations intra case, the cases share the same methodology therefore predict the same results and can be replicated literally (Yin, 2014, p. 57) which in this study means using the same coded scripts and tools to launch and analyze the results. As the hardware resources and time is limited the case structure is also limited to having eight workers maximum in one, two, three, four and eight worker configurations. This case structure compassing of two contexts and multiple cases within a context is show on Figure 6 far below that also combines the multiple case structure from Table 5 in Chapter 3.1.2. Fore mentioned chapter also explores construction and restrictions of said case structure.

Data of this case structure embedding said units of analysis form an archive of data that consists of archived runtime data and *physical artifacts* (Yin, 2014, pp. 117-118) that have concluded into a crash. Archived data in this case can be presumed to be an representation of said runtime statistics but must still be checked for discrepancies as it is easy to be blinded by precise quantitative numbers (Yin, 2014, p. 109). Crashes can be considered physical artifacts because they provide *insight* (Yin, 2014, p. 106) on fuzzing operation as their occurrence is a direct results from crashing input. Crashes are also considered proof of a possible bug and have been used as a metric to assess fuzzer performance (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2131-2132). Data in this study is not *holistic* rather exhibits distinct boundaries within a case and therefore cannot be pooled among different cases. In other words, cases exhibit the same phenomena but do not share the same environment. This is called *embedded multiple-case* (Yin, 2014, p. 62) study and in an embedded case data is used asses the cases both individually and compared to each other but not pooling the data among different cases.

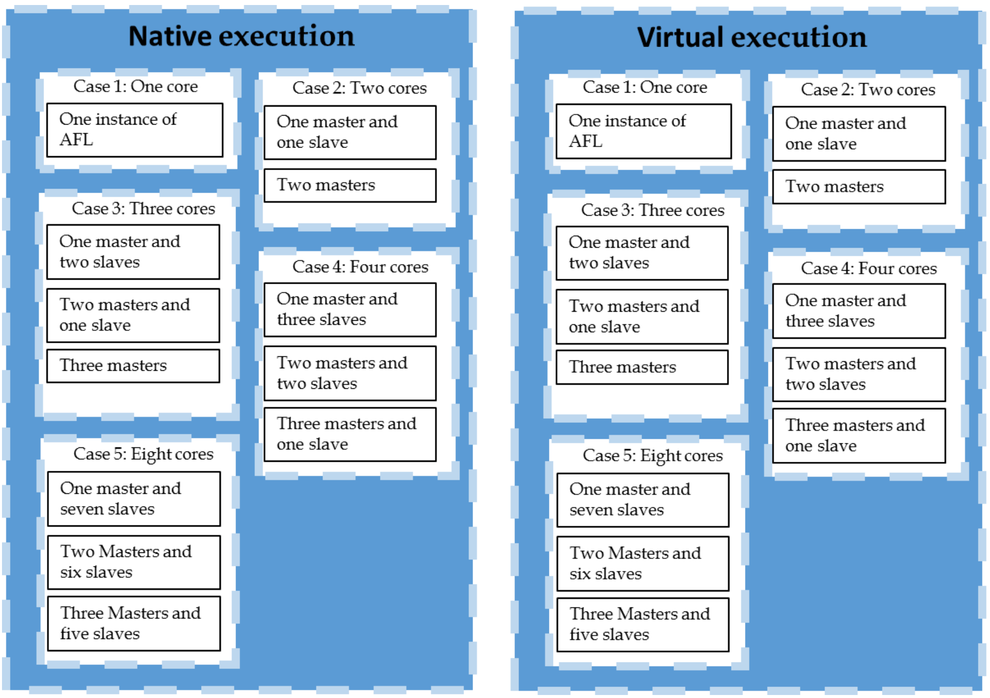

Native execution
Case 1: One core
One instance of AFL
Case 2: Two cores
One master and one slave
Two masters
Case 3: Three cores
One master and two slaves
Two masters and one slave
Three masters
Case 4: Four cores
One master and three slaves
Two masters and two slaves
Three masters and one slave
Case 5: Eight cores
One master and seven slaves
Two Masters and six slaves
Three Masters and five slaves
Virtual execution
Case 1: One core
One instance of AFL
Case 2: Two cores
One master and one slave
Two masters
Case 3: Three cores
One master and two slaves
Two masters and one slave
Three masters
Case 4: Four cores
One master and three slaves
Two masters and two slaves
Three masters and one slave
Case 5: Eight cores
One master and seven slaves
Two Masters and six slaves
Three Masters and five slaves

Figure 6 Case structure of this study

Checking if data within a case is uniform can be done by using statistical methodology, as fuzzers rarely report same results when ran one or twice (Arcuri & Briand, 2011, p. 1). In this study data is linked to propositions using *hypotheses* that are proven or disproven through statistically analyzed metrics of coverage and unique crash count. Additionally, a competing metric is used by assessing each crash (i.e. physical artifact) from different cases in order to find common crashes that can be reviewed if the timing of said crash correlates with other metrics. Both interpreted metrics and linking is done through tabulating the data within a case and comparing this data to each other case. Chapter 3.3 prescribes the statistical methodology used in this study and how it is applied for fuzzer scalability testing.

3.1.2 Multiple-case structure

Multiple-case structure in this study is iteratively constructed knowing the limits of both software being used (i.e. AFL) and hardware availability. In case structure the software limits number of cases as the software consumes (i.e. binds) itself on a single apparent core, therefore this core is not usable to other tasks. Furthermore, as virtualization is used, the CPU cores and threads must not be overburdened, as it might hinder the execution of privileged instructions (namely fork) of used software as discussed in Chapter 2.3.3. Therefore, hardware limits on how many different instances can be run simultaneously, in order not to convolute the platform itself and therefore have negative impact on performance metrics.

Hardware used in this study is two workstation level machines with 2xIntel Xeon E5-2640v4 10 core Hyperthread processors running at 2.4GHz and have 64 GB RAM total per workstation. Workstations ran an Ubuntu 18.04 LTS instance but were re-installed for this experiment. This means, that the whole experiment has a total of 40 cores available in two machines.

In multiple-case studies a *pilot case study* (Yin, 2014, p. 240) was run with UBUNTU 18.04 LTS Desktop (GUI Enabled) and Server branches with 18 cores bound to AFL instances (single instances and deterministic checking enabled). Two cores were left unused as they were determined to be used for Ubuntu Desktop OS applications (Ubuntu, 2020) and therefore omitted from the Server run as the run environment must be close as possible to each other (Yin, 2014, pp. 45-49).

Pilot study confirmed that information from fuzzing instances can be gathered and physical artifacts (i.e. crashes) are saved on drive with enough information to decipher when the crash occurred. Third run of the pilot study was successful and the documentation to on how configuration was altered is encased in Chapter 3.4.4. Results of this successful pilot study of both Desktop and Server branches are tabulated far below (Table 4) with their average speed, crashes, cycles done, and paths explored. Coverage gained was low (sub 3 %) and not all instances found a crash (averaging below 1). Results were gathered from each instance (36 in total) plot data text-file and converted to CSV-file (Comma-Separated Values) and imported to Microsoft Excel.

From this pilot study UBUNTU 18.04 server branch was chosen as it was marginally faster on average (execution speed, Cycles done, and Paths explored) and lack of GUI component did not bother the researcher. Between three runs of pilot study the crash average ranged 1,611 to 5,05 for Desktop and 0,67 to 1,11. Furthermore in Virtual environment virtualizing many Desktop GUIs could be problematic and booting to multi- or single user mode would practically be the same as using server operating system.

Pilot study run three data was also used as the Case 1 data. This is not recommended (Yin, 2014, p. 240). Technically the pilot study was the run one of pilot study and run two was an improvement that finally led on the run three. In this way the run three is identical to all the case study runs, and therefore can be used as a Case 1 in this case. Same Case 1 with same scripts was also ran in Virtual context.

Table 4 Key results of pilot study run three

Metric used	**Desktop**	**Server**
Average exec speed	1431,22	1457,24
Average crashes	1,94	0,78
Average Cycles done	64,94	75,11
Average Paths Explored	182,78	185,37

After confirming that a case could be run on the said hardware and results can be harvested from said cases the multiple case structure had few restrictions. These restrictions were:

- two 18 core hardware resources (reserving 2 cores for OS)
- must be able to run one to three master workers of AFL alongside with several Slave workers
- AFL instances must reside in one of two machines as a whole
- two contexts which are similar in multiple-case structure but cannot be run alongside each other.

AFL instances used in this study are either one, two, three, four or eight core instances. From one to three instances the reasoning is that these in order to test the configuration the master instances must be ran from one two three alongside with their slaves picking up the rest of the cores belonging to this instance. Four-core instance was an addendum to same ideology as it would let this study to explore the first three masters and one slave configuration. As the final instance an eight-core instance was chosen as it exhibits a range of one two seven slaves and is next in binary line-up. According to previous studies (Xu, Kashyap, Min, & Kim, 2017) the scalability should be apparent under ten cores. Therefore speedup should be measurable according to both Amdahl´s- and Gustafsson-Barsis´s law as the amount of combined total parallel results would divide itself among other serial workers according to Amdahl or multiply the results if instance obeys Gustafsson-Barsis (McCool, Robinson, & Reindeer, 2012, pp. 57-65).

With aforementioned logic 12 test cases were identified and are listed above on Table 5 below tabulating number of master and slave instances and cores used. Cases were ran five times for statistical accuracy (Arcuri & Briand, 2011, pp. 4-7) and their scheduling is explained on Chapter 3.2.1. Scheduling was needed as a single run of all cases would not fit inside hardware resources available (36 cores) as apparent by Table 5 (above) last column "Cumulative Core Count".

Table 5 Test Cases

Case name	Masters	Number of Slaves	Cores used	Cumulative Core Count
1-0	1	0	1	1
1-1	1	1	2	3
2-0	2	0	2	5
3-0	3	0	3	8
2-1	2	1	3	11
1-2	1	2	3	14
3-1	3	1	4	18
2-2	2	2	4	22
1-3	1	3	4	24
3-5	3	5	8	32
2-6	2	6	8	40
1-7	1	7	8	48

3.1.3 Configuration of experimental setup

Contemporary events can be observed and manipulated in a case study as events can be observed and directed systematically in an properly restricted experimental setup (Yin, 2014, p. 12). Experimental research has several different pitfalls that must be avoided. Main theory for assuring reliability to performance testing can be done via having strict and precise rules for conducting experimental research (Srinagesh, 2006, pp. 163-165). The experiment is heavily influenced by the research design, hardware and chosen software from Chapter 3.1.1 and 3.1.2. From research design following restrictions can be derived:

- Operation system UBUNTU 18.04 LTS Server
- Virtual and native execution environments
- Fuzzer AFL
- Target LAVA-M who

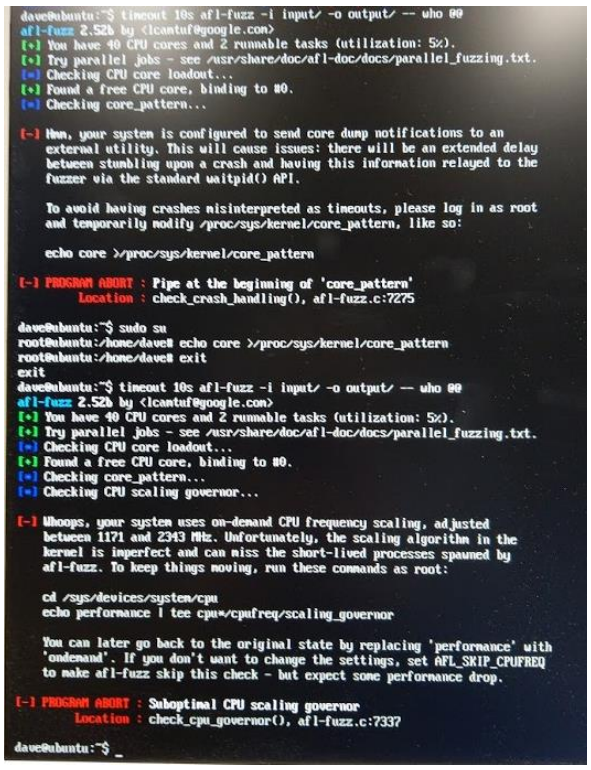
In addition to research design factors the environmental factors of both physical and software must be included by having the same *experimental setup* (Srinagesh, 2006, pp. 56-57) on both native and virtual contexts. Environmental factors are controlled by placing the hardware in an well ventilated office space that is temperature and humidity controlled. Software factors are controlled by having the same OS on both virtual and native execution and removing unnecessary services from said OS during installation. This is done in order to offload random workloads from the OS environment as fuzzing with AFL is an CPU bound task therefore all other tasks in the same operating environment must be kept to minimum in order to increase the reliability of reported data (Srinagesh, 2006, pp. 57-58). Hardware was connected to each other by 100mbit Ethernet switch and an simple private address space was configured to both network interfaces. This was done in order to enable using both workstations through secure shell (SSH) but was also needed in virtualization platform ProxMox discussed in Chapter 2.3.3.

Native environment installation was done with DVD installation disk. Only ssh-server package was installed during server installation as *secure shell* (SSH) as it was used to connect to OS environment in both virtual and native context. ProxMox was chosen as the virtualization platform because it is as per Chapter 2.3.3 a HAL assisted full virtualization system using KVM and free to use. ProxMox version 5.2 was installed from DVD and its web GUI was used to create virtual machines.

Because of context of this study (offline approach and scalability evaluation) from Chapter 1.1 no direct Internet connection was used. Needed packages were smuggled in with USB and local repository was made by first installing the Debian package (abbreviated dpkg) development tools manually with dependencies then reading all the other packages with dpkg scan packages-tool. After that apt sources list was modified to include said package listing (in compressed form) and packages were installed normally with apt-get install as installing the packages with dpkg requires installing dependencies manually (labor intensive) and apt installs them automatically.

Virtualization needs VM profiles that have to match the test case structure of table 5 from Chapter 3.1.2 and therefore one VM was made first with the same configuration that off the native execution machine from above paragraph and then cloned and its CPU count modified to match the scheduling of multiple cases. Virtual machines were configured with 2 gigabytes of RAM and 20 gigabyte dynamically allocated hard drives. These resources are sufficient for Ubuntu server installation, which requires 512mb of RAM and 15 gigabytes of non-volatile memory (Ubuntu, 2020).

AFL will be limited to version 2.52b, which was the latest version during experimental phase in during first half of 2019 and available through apt. LAVA-M was downloaded from github (Dolan-Gavitt, et al., 2016) and configured to use AFL-specific compilers to instrument the target program who. During testing the installation AFL produced a warning from CPU speed governor - and core dump settings which were reconfigured according to AFL printout as shown below on Figure 7. This configuration was checked during start-up of scheduled instances by scripting an exit and error message if any warnings manifest (see Appendix 1 for details).



```
dave@ubuntu:~$ timeout 10s afl-fuzz -i input/ -o output/ -- who @@
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 40 CPU cores and 2 runnable tasks (utilization: 5%).
[+] Try parallel jobs - see /usr/share/doc/afl-doc/docs/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...

[-] Hmm, your system is configured to send core dump notifications to an
    external utility. This will cause issues: there will be an extended delay
    between stumbling upon a crash and having this information relayed to the
    fuzzer via the standard waitpid() API.

    To avoid having crashes misinterpreted as timeouts, please log in as root
    and temporarily modify /proc/sys/kernel/core_pattern, like so:

    echo core >/proc/sys/kernel/core_pattern

[-] PROGRAM ABORT : Pipe at the beginning of 'core_pattern'
         Location : check_crash_handling(), afl-fuzz.c:7275

dave@ubuntu:~$ sudo su
root@ubuntu:/home/dave# echo core >/proc/sys/kernel/core_pattern
root@ubuntu:/home/dave# exit
exit
dave@ubuntu:~$ timeout 10s afl-fuzz -i input/ -o output/ -- who @@
afl-fuzz 2.52b by <lcamtuf@google.com>
[+] You have 40 CPU cores and 2 runnable tasks (utilization: 5%).
[+] Try parallel jobs - see /usr/share/doc/afl-doc/docs/parallel_fuzzing.txt.
[*] Checking CPU core loadout...
[+] Found a free CPU core, binding to #0.
[*] Checking core_pattern...
[*] Checking CPU scaling governor...

[-] Whoops, your system uses on-demand CPU frequency scaling, adjusted
    between 1171 and 2343 MHz. Unfortunately, the scaling algorithm in the
    kernel is imperfect and can miss the short-lived processes spawned by
    afl-fuzz. To keep things moving, run these commands as root:

    cd /sys/devices/system/cpu
    echo performance | tee cpu*/cpufreq/scaling_governor

    You can later go back to the original state by replacing 'performance' with
    'ondemand'. If you don't want to change the settings, set AFL_SKIP_CPUFREQ
    to make afl-fuzz skip this check - but expect some performance drop.

[-] PROGRAM ABORT : Suboptimal CPU scaling governor
         Location : check_cpu_governor(), afl-fuzz.c:7337

dave@ubuntu:~$ _
```

Figure 7 AFL CPU speed governor and core dump setting warning printout and fix.

3.2 Data collection procedures

Data collection procedures in this study are formulated in this chapter. In order to use the hardware resources of this study effectively the cases and units-of-analysis must be scheduled as they use more resources that are available. Data for this study is gathered for four metrics: unique bugs found, code coverage, execution speed and shared bugs.

3.2.1 Scheduling and launching the multiple-case structure

As demonstrated on Chapter 3.1.2 the chosen cases of this study do not fit inside the hardware resources available. Easiest solution would have been to run the different cases serially back to back, but hardware utilization would be uneven as the cases would not fit inside a single nor both machines. Furthermore the cases must be run multiple times (Arcuri & Briand, 2011, pp. 4-7) (Klees, Ruef, Cooper, Wei, & Hicks, 2018, p. 2124) which concludes that the cases must have schedule that uses at maximum 18 cores from both workstations and handles the multiplication of test runs.

First multiplication of test cases that uses all the cores and handles all the cases is three multiples of all cases but this is deemed the bare minimum of multiplication for accuracy (Arcuri & Briand, 2011, pp. 4-7). The next multiple of runs is six but the more runs are included the harder scheduling comes. Multiplication of runs also consumes more time and with optimal scheduling five multiplications would take 16 days including both Case 0 but with six multiplications it would consume 18 days. Five was chosen as it proved to be easier to schedule than six.

Scheduling of cases is presented on Table 6 below which shows what cases, and which run of a case is being run on which day of experimenting on which workstation. Case name is derived from the number of masters with first number, number of slaves with second number and multiple of case run with "r" and number after it. This naming convention was also used when launching the instances. Sub optimal core usage can be seen from Day 5 onward but as AFL is a CPU bound task the sub optimal usage only affects the usage of common resources (storage, memory and network) which are abundant during execution as the only processes executing are the OS and AFL.

Scheduling of instances was done with BASH-scripts which are added to Appendix 1. First script of Case 0 launched AFL 18 times with enough time (2 seconds) between them to circumvent the problems mentioned in Chapter 3.4.1. Second script for "multimaster" application was developed to construct case file structure and launch and close user supplied amount of master and slave AFL instances all pointing to required single *sync dir* (AFL synchronization directory for parallel fuzzing (Google LLC, 2017)). Closing of a "multimaster" launched instances was done by command timeout with supplied parameter of 24h which closes the instance after 24 hours of uptime.

Table 6 Scheduling of test cases

# Day	Work-station	Cases	Used Cores
Day 1	1	Case 0 study: 18 single AFLs UBUNTU Server	18
Day 1	2	Case 0 study: 18 single AFLs UBUNTU Desktop	18
Day 2	1	35r1, 35r2,11r1	18
Day 2	2	35r3, 35r4, 11r2	18
Day 3	1	35r5, 31r1,31r2,11r3	18
Day 3	2	17r1, 31r3,31r4,11r4	18
Day 4	1	17r2, 17r3, 11r5	18
Day 4	2	17r4, 17r5, 20r1	18
Day 5	1	20r2,20r3,31r5,12r1,12r2,12r3	17
Day 5	2	20r4, 20r5, 22r1, 21r1,21r2,21r3	17
Day 6	1	12r4,12r5,26r1,13r1	17
Day 6	2	21r4, 21r5, 26r2, 13r2	17
Day 7	1	26r3, 13r3,30r1, 30r2	18
Day 7	2	26r4, 13r3,30r3, 30r4	18
Day 8	1	26r5, 13r4, 30r5	15
Day 8	2	22r2, 22r3, 22r4, 22r5	16

Launching multiple instances was done through screen. This utility program multiplexes between several terminals (Laumann, Davison, Weigert, & Schroeder, 2019) and allows the user to specify command to be run inside said multiplexed terminal. Therefore the "multimaster" BASH-script was set to automatically run after launching a new screen to other terminal leaving the current SSH session terminal free for launching other scripts or checking the status of screen. Multiple cases were run on single workstation (in native execution context) by calling the "multimaster" script after checking that the previous run was complete through checking that screen listing (screen -ls) was empty. This process is show on Figure 8 (far below) which shows the order of BASH-scripts and their functions as a flow diagram.

For the virtualized context the machines were cloned from an *template* which was created the same way as the native context UBUNTU machine. Only addition to this workflow that of cloning the different machines and changing their CPU count before launching the VM. This addition is an addendum to Figure 8 below and the process continues the same way as in native execution environment.

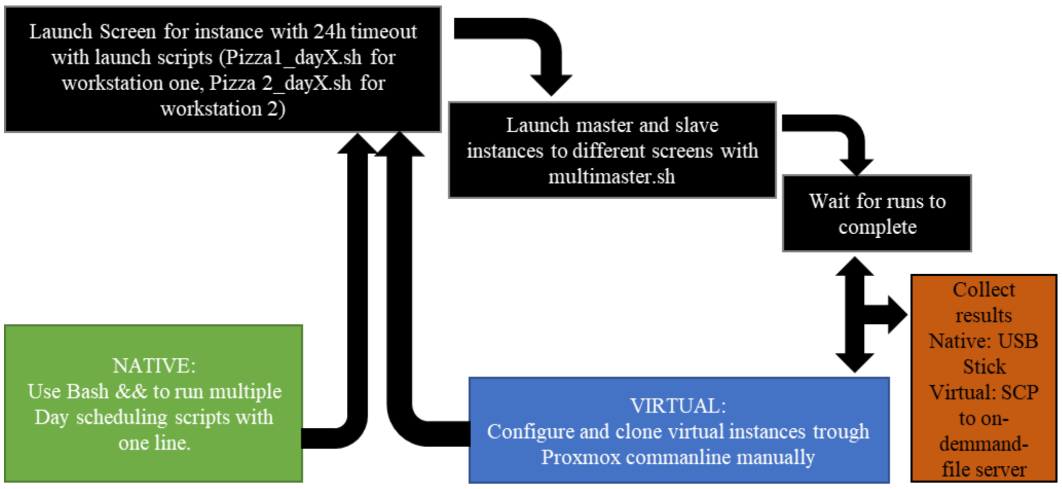

Launch Screen for instance with 24h timeout with launch scripts (Pizza1_dayX.sh for workstation one, Pizza 2_dayX.sh for workstation 2)
Launch master and slave instances to different screens with multimaster.sh
Wait for runs to complete
Collect results Native: USB Stick Virtual: SCP to on-demmand-file server
NATIVE: Use Bash && to run multiple Day scheduling scripts with one line.
VIRTUAL: Configure and clone virtual instances trough Proxmox commanline manually

Figure 8 Process of launching AFL through scripting

Data of instances (i.e. *sync dir*) was copied to and USB stick and transported for further analysis. This data was copied after every run and verified that data structure observed was the same as that was scheduled. One anomaly was found after all data was gathered from native execution context as Day 5 of workstation 1 scheduling script called the "multimaster" with wrong parameters leading to rerun of said day. Process of finding this anomaly and how it affected the validity and reliability of this study is explained on Chapter 3.4.2. In virtual context the data had to be transported through sending the data with scp to a centralized file server VM which was purposefully created from the same clone and kept shutdown during experiments. After the data was collected from all virtual context cases the whole VM was exported and its data excavated for data analysis to Windows machine.

3.2.2 Unique bugs found

According to Klees & al. there are three main ways of deduplicate bugs: assessing them through bug behavior (against *ground truth*), assessing it against *coverage profile* and *stack hashes*. Both *ground truth* and *coverage profile* deduplicate the crash by adding an labor intensive checking phase where bugs are assessed against software fixes (or by patching the bug by themselves). *Stack hash* can be done automatically, but might not be that accurate. (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2131-2135)

Ground truth can be established if we know for certain that a specific bug is triggered by an specific input which usually happens with old known bug and input tuples or synthetically injected bugs. Therefore, the target must have an good patch history and the patches must be available through repositories. Other option is to use synthetic benchmarks (i.e. LAVA, CGC 16, see Chapter 2.1.5) which is possible in this study but could still requires the bugs to exhibit unique behavior when crashing. (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2131-2132)

Coverage profile is defined by using *taint tracing* or dynamic checking during crashes. This can be done by either using an *taint inducing framework* (like Valgrind from Chapter 2.1.1) or by using different sanitizers (for example *ASAN* Address SANitizer or *UBSAN* Undefined Behavior SANitizer) to dynamically assess used memory and its state leading to crash and during crashing (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2132-2133). Using these methods could still lead to imprecise crash deduplication and might if target is propped up with additional resource burden (*taint tracing*) could require to re-think hardware resource usage in this case study.

Stack hash relies on program execution order, more precisely it uses information from *call stack* to determine what *frames* the crashing input made through before crashing the program. When the program crashes *n stack frames* (usually 3 to 5) are examined and hashed together to form the *stack hash*. Therefore, if the input travels different frames (therefore a different *program path*) the hash itself is always different. On the other hand, if only the value passed to frame changes, but path remains the same the *stack hash* will not change which indicates that the path taken is not unique. Stack hashing is also known to under- and over count bugs. This might be due to the crash having a long set path that is triggered early and passes through several frames before crashing or if the path itself is somehow randomized based on passed value. Nevertheless stack hashing has proven itself being more precise than AFL coverage profile and it can be done after the crashes have been gathered which can be done after the fuzzer has completed its work. (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2133-2134)

AFL´s bug detector is known to over-inflate bugs when comparing to ground truth (Van Tonder, Kotheimer, & Le Goues, 2018, p. Table 1) (Klees, Ruef, Cooper, Wei, & Hicks, 2018, p. 2133) and therefore stack hashes are used to filter bugs. Stack hashing is done by checking crashing inputs with *Crashwalk* software-suite (Nagy B. , 2018) which uses the faulting frame, contents of stack and registers to do the *stack hash*. Crashwalk supports natively AFL-format crashing inputs.

Crashes are deduplicated trough the said *stack hash* and their exploitability is examined automatically (by Crashwalk) with EXPLOITABLE! GDB -plugin but the result whenever the bug is exploitable or not is not considered usable data as it is not performance related rather more qualitative in nature.

3.2.3 Code coverage

AFL measures coverage gained by examining program state transitions during crashing input. Therefore, AFL considers new coverage gained by finding a crashing input that explores target program. This kind of deduplication method is not without flaws as program logic can lead to overcounting unique bugs in the program and therefore inflating the crash count. (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2132-2133)

Van Tonder, Kotheimer and Goues present this kind of overinflating of bugs in their article Semantic Crash Bucketing. In this article AFL has found an bug in a database program (*sqlite*) search function (SELECT … FROM) that with manipulation of said search in crash inducing but meaningful way returns a new unique crash as the path to crash traverses trough different parts of program. In an effort to combat this Van Tonder, Kotheimer and Goues propose a technique of *program transformation* (i.e. *binary patching*) to patch program for crashing inputs and then checking if the crashing input still crashes the program. If no crash occurs with said crashing inputs it can be said that previously crashing inputs that now failed can be clustered together. If and input crashes through the patch it is traced and fixed and checked again to be clustered with same behavior crashes. (Van Tonder, Kotheimer, & Le Goues, 2018, pp. 2-3)

Likewise, coverage can be seen as a function of generated inputs and therefore there is no certainty that generated inputs reach full coverage of the program i.e. the program might have *unreachable code* which cannot be reached during normal execution of program. Without knowing the exact number of bugs, the results of coverage gained is always uncertain. Likewise there is no fundamental reason to believe that maximizing code coverage yields more crashes (Klees, Ruef, Cooper, Wei, & Hicks, 2018, p. 2135). (Arcuri & Briand, 2011, pp. 5-6)

As it is know that AFL-reported coverage might not be truly accurate (as paraphrased above) unique bugs deduced with methodology detailed in Chapter 3.2.2 are used alongside to determine if the numerical value of coverage reported correlate with number of unique bugs found. As an example: if two instances report and coverage of 10% but second instance reports and is confirmed to have three more unique crashes the second instance has gained more coverage although AFL reports the same amount of coverage from both instances. In this case the second instance has managed to find new crashes, but the first instance has found an crash that with small variation to crashing input produces new coverage according to AFL, the same situation as represented on second paragraph of this chapter. In this study coverage reported by AFL is used as a starting point for coverage metric and it is compared with unique bugs for an metric to compare different cases and configurations to each other.

3.2.4 Execution speed

Xu et al. have used execution speed as an scalability metric in their study of "Designing New Operating Primitives to Improve Fuzzing Performance". In this study scalability of AFL different components is examined and found lacking. This study proposes many improvements to AFL design and also show the improvements by testing AFL and LibFuzzer against Xu et al. derivative AFL^{OPT}. (Xu, Kashyap, Min, & Kim, 2017)

Testing itself is not done up to any standard that would fulfill the requisites laid out by Klees & al or by Arcuri & Briand. The execution time of each test is 5 minutes (Xu, Kashyap, Min, & Kim, 2017, p. 11) where all fore mentioned authors prefer longer runs of more than 24 hours to have more statistical accuracy for experiments (Arcuri & Briand, 2011, pp. 7-8) (Klees, Ruef, Cooper, Wei, & Hicks, 2018, p. 2126). Furthermore, the study does not explicitly say how many times the tests were run. Reasoning for creating these short and non-replicative tests is not presented.

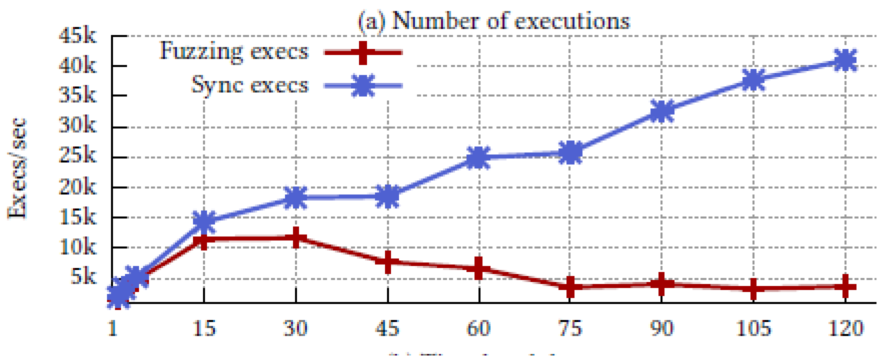


Figure 9 Multicore scalability of AFL according to Xu et al. (Xu, Kashyap, Min, & Kim, 2017, p. 4)

While it is shown that new operating principles applied in aforementioned study enhance the performance during short experiments (Figure 9 above) on the long-term effects are not explored. The mechanic of execution speed is interesting to test if it can be applied to measure speedup with multiple workers. Therefore, in this study the execution speed is included as a metric to assess if it can be used as a metric to measure performance differences between master and slave instances of AFL.

3.2.5 Shared bugs

It is highly likely that the instances will most probably explore multitudes of same crashes as they are always started the same way. Therefore, they share bugs between instances and some instances with more computing power might be faster to find these bugs. If this is the case, the speedup can be measured and therefore scalability determined from this metric of checking how long it takes for an instance to find a bug that is shared between multiple instances.

Speedup (as defined in Chapter 2.2.2 by Equation 1) is measured from dividing latency of one worker to the latency of multiple workers. Latency is defined in Chapter 2.2.1 as time taken for to complete a task. Therefore, we can safely say, that a fuzzers work is to produce crashes. The crash must have a timestamp form which we can determine how long has it taken for an fuzzer to produce said crash.

AFL provides the timestamp for starting the instance and it also saves the *physical artifacts* (including when it was created i.e. found) of said crashes. From this information we can determine how much *latency* there is per crash and after deduplication we can determine the *latency* of a bug. Therefore, it is possible to determine the speedup of instances if an shared bug between instances is found. It is most likely, that this is a *shallow bug* and therefore it would be better if multiple shared bugs are found in order to have more data points for comparison.

3.3 Guidelines for data processing

3.3.1 Analytical strategy

Knowing that the data points available are already well labeled there is probably no need to start tabulating or putting them in different arrays or graphs in order to grasp how the data points correlate within an embedded analysis unit. It is however very much mandatory to graph (and therefore tabulate) the results from different analysis units in order to be able to measure the difference in metrics. These data points are gathered the same way from every instance in order to retain validity in data gathering.

The strategy itself to process the data is grounded on *theoretical proposition* of performance theory from which we examine the scalability and if it follows Amdahl´s or Gustafsson-Barsis´s law. As we have two theories competing each other we can say that they *plausible rivals* to each other. This mandates that we must have theory driven metrics that can be used to craft rival *hypotheses*. (Yin, 2014, pp. 136-142)

In this study in order be able to analyze the data it must be in a form where we can apply theoretical context of speedup and scalability to it. Therefore, the rivals are automatically formed between cases as we can assume that adding more workers effects the speedup of an AFL instance. In the same sense we can form common hypothesizes that work between all cases as we only need to measure speedup of said instance compared to speedup of other larger or smaller instance in order to be able to decide if any speedup has occurred. These common hypnotizes are constructed on Chapter 3.3.2.

Because of random nature of fuzzing it is bad practice to just compare single runs to each other (Klees, Ruef, Cooper, Wei, & Hicks, 2018, pp. 2127-2128) which lead to scheduling multiple runs across many days as Chapter 3.2.1 dictates. Using statistical methodology and namely Mann Whitney U-test to make a decision about proving or disproving hypotheses is to be used according to Arcuri and Briand (Arcuri & Briand, 2011, pp. 3-5) and is detailed in Chapter 3.3.2. This must reflect in common hypotheses on Chapter 3.3.3.

3.3.2 Statistical methodology

Statistical test used in this study is Mann-Whitney U-test. It is used because randomized algorithms do not follow normal distribution that is required in more popular Mann-Whitney T-test and Welch test. Therefore we must first know how the metrics of Unique bugs found, Code Coverage, Execution speed and Shared bugs are ranked in order to be able to use Mann-Whitney U-test (Arcuri & Briand, 2011, pp. 4-5,8).

In order to use Mann-Whitney U-test we must establish ranking order to our metrics. For Unique bugs and Code Coverage we can average the metric in order to have some understanding if all instances have rough metric changes but ultimately we can just use max values of each run to have an *n*-size of five for each comparable case in both contexts. Execution speed on the other hand must be averaged before using it in Mann-Whitney U-test. Execution speed is a metric that is already a estimation of speed along the whole running time and therefore we cannot just use max values from a certain run as it most likely has almost the same *variance* between runs. As an concept the shared bugs already report the minimum amount of time used to find a shared bug therefore that metric can be used in Mann-Whitney U-test to form ranking on both comparisons.

After successfully reporting the p-values from Mann-Whitney U-test and passed discretionary judgement (Yin, 2014, p. 61) if the hypothesis is correct or not we can measure with Vargha and Delaneys $\hat{A}_{12}$ to decipher the probability of higher core count AFL instance to produce better results. Measuring this *effect size* and reporting its p-value is also highly recommended (Arcuri & Briand, 2011, pp. 6-7). I have collected the measured metrics and their used values for Mann-Whitney U-test and $\hat{A}_{12}$ to Table 7 below and added what else metrics are reported because of validity and reliability (Arcuri & Briand, 2011, p. 8). As the results are not normal distributed variance and min-values are not reported.

Table 7 Used metrics and applied analysis

Metric	Measurement used for U-test and $\hat{A}_{12}$	Metrics reported
Unique bugs	Deduplicated max value	*p* values, mean, median, standard deviation and max value
Code Coverage	Max value	
Execution speed	Average from a run	
Shared bugs	Min value from a run	

3.3.3 Common hypotheses

In statistical testing a *null hypothesis h0* is usually crafted to use statistical testing to decide if there is statistical difference between results of object A and B. From this test a value of p is calculated and the closer it is to zero the more probable it is for A to be statistically superior to B. Normally a *significance level* of α=0,05 is used to decide this in natural sciences. (Arcuri & Briand, 2011, pp. 3-4)

In case study this kind of discreet decision making must be avoided. The decision where *A* is better than *B* must also include other measures taken either in statistical sense or other to support decision making process. Therefore significance level of α=0,05 is only used as a guideline and values are reported as is to have more open approach in deciding if *A* is better than *B*. (Yin, 2014, p. 61)

Inside a single case we must test multiple times if one configuration is better than the other as they have small configuration changes that might have different performance. With this kind of testing we can directly have answer to RQ3 ("How does the performance differ when using a different number of master and slave configurations in AFL?"). As we have three different configurations, we need three tests: one for each master configuration to be tested against other configurations. This testing must also include both contexts as results from this test are used to answer RQ2 and RQ1. As Mann-Whitney U-test is a comparison of ranks between two groups we can form a *null hypothesis* that states that there is no difference in configurations:

h30: There is no difference in performance
h31: One-master configuration does perform better than other two configurations
h32: Two-master configuration does perform better than other two configurations
h33: Three-master configuration does perform better than other two configurations

RQ2 on virtualization can be directly compared case to case as it mainly focuses on determining if there is any statistical difference in performance. From this we can pairwise test if one instances results are better than others. This leads to a *null hypothesis* of "there is no difference in contexts" and following sub-hypotheses, where we denote native execution as "N" and virtual execution as "V":

h20: There is no difference in performance considering N and V executions
h21: There is no difference in performance in contexts in case one
h22: There is no difference in performance in contexts in case two
h23: There is no difference in performance in contexts in case three
h24: There is no difference in performance in contexts in case four
h25: There is no difference in performance in contexts in case five

In this study the goal is to conclude if there is speedup gained by using more AFL workers (RQ 1). This can be done by using the metrics from case 1 and all other cases to compare them to all pairwise for statistical difference. Pairwise comparison is used as speedup is the function of latency of one worker (i.e. case 1) to multiple workers (other cases). From this we can conclude if speedup has occurred and do not need to test speedup between cases 2 and 3 for example. If there is no apparent speedup (i.e. *p* value of U-test stays above 0,05) it is possible still to test if the eight worker instance is statistically better than the one core instance as they are the furthest away of each other when hardware resources are considered. This leads to following hypotheses:

h10: There is no statistical difference in instance performance
h11: Two-core instance is better than one core instance
h12: Three-core instance is better than one core instance
h13: Four-core instance is better than one core instance
h14: Eight-core instance is better than one core instance

3.4 Validity

Validity of this study is examined though the construction of case research design, internally providing explanation on how decisions are made and externally validating the results through previously done research. During the experimental phase few threats were identified concerning pilot study and data gathering, both of which have been detailed in this chapter. No major threats to validity of this research were identified.

3.4.1 Construct validity

In a case study the construction of research design is validated by using multiple sources of evidence, establishing a chain of evidence and by reviewing the case study report. This is done in order to validate and identify correct operational measures for the phenomena being studied. In order to meet this validation two steps must be covered (Yin, 2014, pp. 45-46):

1. Define case study objective and its phenomena
2. Identify how these are measured

In this study research design is validated through performance theory namely by measuring the speedup of an AFL instance. From this embedded unit of analysis multiple evidence sources are identified and gathered: Deduplicated crashes, code coverage, execution speed and discovery time of a shared bug. In nature this evidence is both historical and physical artifacts as first three of evidence categories are time-based records of AFL instances statistics and the final evidence is a proof that a crash has occurred.

Evidence is gathered from its context and dissected to excel in order to do data analysis. All available data of fore mentioned metrics are analyzed, and their results are viewed through statistical analysis to decide if any speedup has occurred in any of the metrics. This process is documented on Chapter 3.3.

By accounting for multiple evidence sources and showing how the evidence is collected and analyzed we can decide that construction of this case study is done properly. It is to be noted that these measures might be inadequate if the phenomena in question is not scalable or the metrics are chosen poorly but there is no indication in literature that AFL would not scale with multiple workers (Xu, Kashyap, Min, & Kim, 2017) and the metrics have been used in other studies (Klees, Ruef, Cooper, Wei, & Hicks, 2018).

3.4.2 Internal validity

Internal validity of research design is compromised if not all factors are counted into causal explanations or inference while conducting is not properly investigated. There are multiple ways of determining if internal validity is intact by doing *pattern matching, explanation building, addressing rival explanation* and by using *logic models*. In this kind of experimental study, the internal validity is great concern. (Yin, 2014, pp. 47-48)

In this study the main tool for validating the research design internally is to use pattern matching to match the gained data points in their metric to Amdahl's or Gustafson -Barsis law (see Chapter 2.2.3). This can be done by plotting each metric both inside the case and between cases. If the metric shows linearly or exponentially growing (with historical data) or declining (as with shared bugs as lower time is better) results it can be said that it most likely scales according to either law. It is of course possible that metrics do not yield any scalability information, or the experiment is too short or not replicated enough to provide enough information.

Pattern matching in this study is enhanced by using Amdahl and Gustafson-Barsis law as rival explanations on scalability. Amdahl's law is used as an example of weak scalability where adding more workers will not show increase on latency and can show increase in latency. Rival of weak scalability is strong scalability which offers less latency for workers as they are added.

Explanation building is a subset of pattern matching where causality of variables are used to explain "how" and "why" something happened (Yin, 2014, pp. 147-148). In this study we cannot explicitly say how and why a crash has occurred as the algorithm is random in nature on how it proceeds. Therefore, explanation building is not used in this study to fortify internal validity.

For the same reason of randomized behavior of fuzzing logic models cannot be used to explain how an decision is made as adding a scribe to keep track of state transitions would most likely cause too much inference (and most notably more performance issues) to the fuzzer itself and therefore would be an liability to other record keeping.

3.4.3 External validity

External validity of research design contemplates the generalization of a case study (Yin, 2014, p. 48). External validity is achieved by having literature-derived metrics (see Chapter 3.2) and a theoretical framework for assessing said metrics. Threats to external validity are chosen target (LAVA-M) and fuzzer (AFL) as they are very through studied in fuzzer literature. However, these threats most presumably do not manifest a risk for this study as the we ask the question of "How" does AFL scale rather than try compare AFL to other fuzzers. Furthermore, AFL is very widespread in both its derivatives (Klees, Ruef, Cooper, Wei, & Hicks, 2018) and usage (Chapter 2.1.5 Table 2). Therefore, the general knowledge of how AFL scales is very much generalizable to form a consensus of how AFL should be used in parallel. Therefore, the results of this are externally valid.

3.4.4 Threat to internal validity: Pilot study iteration

In the first iteration of pilot study from Chapter 3.1.2 did not launch the fuzzer instances fast enough (same kind of problem which was encountered in launching 256 instances of AFL (Gamozo Labs, 2018). This was apparent on averaging the execution speed of all instances and comparing them to other instances: a good instance averaged around 1400 executions per minute for Desktop and 1300 per minute for Server Ubuntu branch. This was not common as only one instance out of 18 exhibited this behavior across all pilot study instances, but this kind of behavior was not expected, therefore pilot study had to be modified in order to eliminate this.

Modification to the pilot study was made by lifting the disk operation of reading the input file of LAVA-M to *tmpfs* (temporary file system, RAM disk) which would yield more operation speed compared to normal disk read (Hard Disk Drives ,HDD , in this case). This did not provide better results, and actually worsened them by failing one instance on Desktop side and none on Server side. Although the read of input file was made easier by using more hard drive I/O bandwidth the results were still too volatile to be used in this study. For the third iteration the launching script was modified to wait for 5 seconds (`sleep 5s`) before launching. After the run was complete, this anomalous behavior was not apparent and therefore the third script was used as base to launch multiple worker instances.

3.4.5 Threat to internal validity: Two anomalies during experimental phase

After collecting the data from native execution, it was discovered that Day 5 data was 31r5 case folder was missing. After investigating that all other data was present and accounted for the culprit was found on Day 5 Script (Check Appendix 1 for details). Instead of running a 3 master 1 slave instance for fifth time it ran 1 master three slave instance for fifth time. In order to remedy this the offending cases were investigated, but found intact, as the file structure was already made AFL would not overwrite it as it found an "sync dir" inside its output directory. Therefore, day5 script was rerun with corrections and its results harvested.

During analyzing the results, it became apparent that 31r4 from virtual execution was missing. The study had continued to a point where the hardware resources were no longer available and the analyzes of native execution were done. We deemed it unnecessary to rerun the case, as U-test and $\hat{A}_{12}$ work with uneven population sizes. Therefore, the virtual case 31r4 is missing and replaced with run average on figures while statistical tests follow the normal procedure.

3.5 Reproducibility and Reliability

Reproducibility of this research relies on documenting the process and used tools. Reproducibility is achieved by using scripting language as tool to launch processes identically for all fuzzer instances. Reliability is achieved by diligent data gathering and by evaluating the answers that are presented through this study.

3.5.1 Reproducibility

Reproducing of this study is made possible by using BASH scripting and reporting used hardware and software configuration. BASH scripts are included in Appendix 1. While the scripts themselves are not up to programming standards they are still usable and will reproduce the results when needed.

As we are using scripts and strict case-structure with commonly known methodology of statistical analysis the changes of hardware will change the metric values. Future hardware might do parallelism without the OS or its applications knowing. Therefore, it is imperative to force CPU binding and disable the throttling mechanism if this study's methodology is used.

It is possible that not all aspects of this study have made it into this thesis. I have included every bit of data available to me as researcher but having to translate everything from native language to English is bound to lose something in translation. To avert this, I have re-checked my writing multiple times and even used fellow students to evaluate my work and ask if something is explained poorly.

3.5.2 Reliability of data gathering process

Reliability of data gathering relies that all scripts have executed properly, and the results are gathered from workstations before they are used for other purposes. The most critical phase of gathering data is as the data is transferred from a CSV form to Excel spreadsheet as it is the phase where data might get lost. Therefore, each CSV is imported and checked to be valid in Excel as well as in CSV form.

Data analysis is done with Excel and therefore it is bound to have *formulas* that link to a certain data set (for example counting averages). Formulas are deceitful as they do count the results of any given area. Therefore, when using formulas to analyze the data an extra care must be taken that formulas do not point to the wrong data set.

As explained in Chapter 3.3.2 the p-values of Mann-Whitney U-test are reported. These values are accompanied by effect size calculations from $\hat{A}_{12}$. All of the above hypotheses must be tested with all claimed metrics from Chapter 3.3.2 Table 7. It is possible that there is no statistical difference between instances in which case the RQ1 can still be answered through calculating the speedup for each instance for their minimum and maximum value and plotting this graph to observer skewness of the graph. If no skewness is present either the metrics are not correct for performance testing or the experiment phase of this study has failed.

4 Results

Results of this study are twofold: Individual instance results are recorded on Appendix 2-3 and are analyzed in this chapter. All data was gathered to an Excel worksheet trough using Linux utility program `ssconvert` which can among other things convert multiple CSV files to excel files. Data was then re-ordered for each metric and reported here on following chapters 4.1 – 4.5.

4.1 Code coverage

Code coverage was measured from AFL instance provided data .CSV files. From this the Maximum value for an instance was used as the coverage gained for the instance and displayed in table 8 below. In this table the native and virtual contexts are side by side and their instances and reruns maximum coverage displayed as percentage of target programs codebase.

Table 8 Coverage gained in both contexts

	Coverage gained (%) in native execution environment					**Coverage gained (%) in virtual execution environment**				
	r1	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
1m1s	2,19	2,17	2,18	2,18	2,18	2,19	2,19	2,19	2,19	2,18
2m	2,17	2,18	2,18	2,17	2,18	2,18	2,19	2,19	2,18	2,19
1m2s	2,18	2,19	2,19	2,19	2,18	2,19	2,19	2,19	2,19	2,19
2m1s	2,17	2,18	2,18	2,17	2,17	2,19	2,19	2,19	2,19	2,19
3m	2,18	2,18	2,18	2,17	2,18	2,18	2,18	2,18	2,19	2,18
1m3s	2,19	2,17	2,19	2,18	2,19	2,19	2,19	2,19	2,18	2,19
2m2s	2,17	2,17	2,17	2,17	2,17	2,19	2,19	2,19	2,19	2,19
3m1s	2,19	2,18	2,17	2,18	2,18	2,19	2,19	2,19		2,19
1m7s	2,18	2,19	2,19	2,18	2,18	2,19	2,19	2,19	2,19	2,19
2m6s	2,19	2,18	2,19	2,18	2,19	2,19	2,19	2,19	2,19	2,19
3m5s	2,19	2,19	2,18	2,17	2,19	2,19	2,19	2,19	2,19	2,19

As can be seen on this table 8 above most of the instances did not venture beyond 2,18 % of the codebase of the program. This is quite low number considering that the AFL did run for 24 hours before quitting. Data from virtual context´s 3 master 1 slave rerun number four is missing. In this case it is not meaningful as all the data is so homogenous. Following table 9 (far below) further analyses the coverage gained in both contexts.

Table 9 Average, median and standard deviation of gained coverages

Native	1m1s	2m	1m2s	2m1s	3m	1m3s	2m2s	3m1s	1m7s	2m6s	3m5s
Avg. cov. %	2,18	2,18	2,19	2,17	2,18	2,18	2,17	2,18	2,18	2,19	2,18
Med. cov %	2,18	2,18	2,19	2,17	2,18	2,19	2,17	2,18	2,18	2,19	2,19
Std. dev. %	0,01	0,00	0,00	0,00	0,00	0,01	0,00	0,01	0,00	0,00	0,01
Virtual											
Avg. cov. %	2,19	2,19	2,19	2,19	2,18	2,19	2,19	1,75	2,19	2,19	2,19
Med. cov %	2,19	2,19	2,19	2,19	2,18	2,19	2,19	2,19	2,19	2,19	2,19
Std. dev. %	0,00	0,00	0,00	0,00	0,00	0,00	0,00	0,88	0,00	0,00	0,00

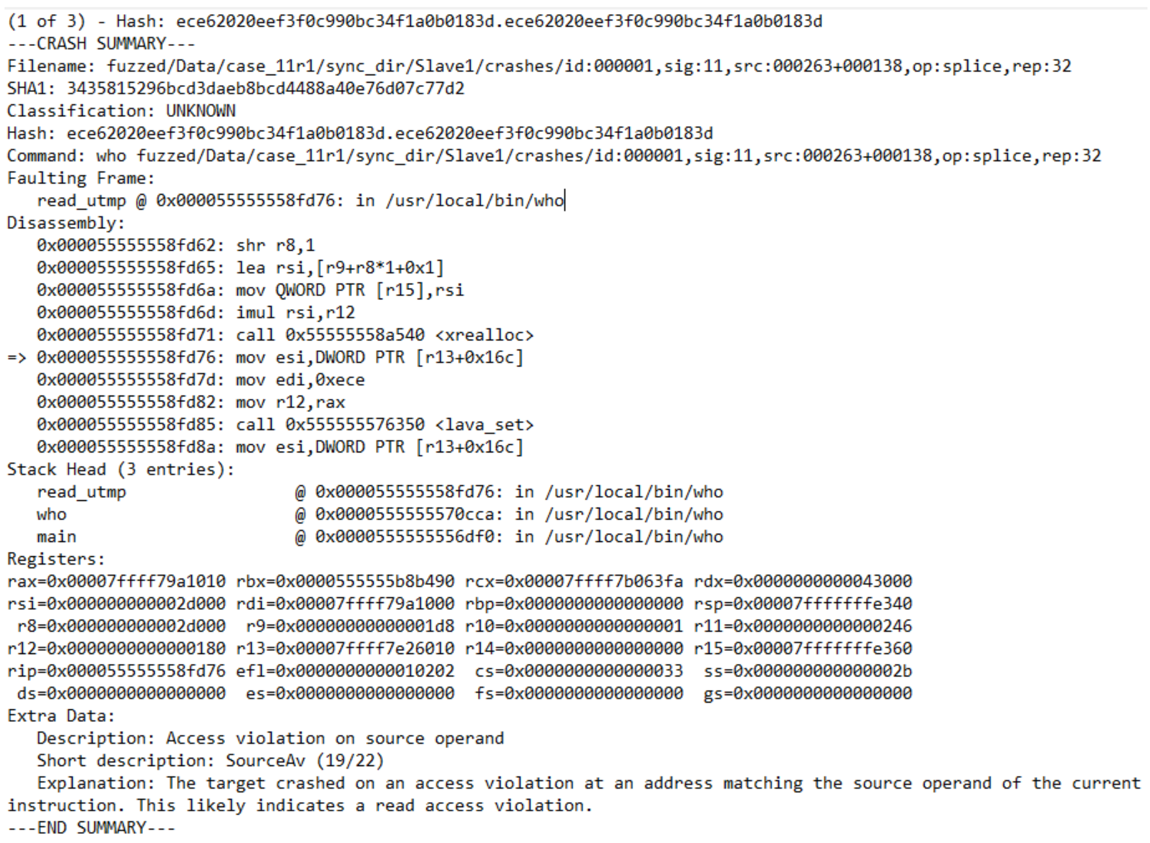
From both tables 8 and 9 (above) it is clear that, in this dataset the coverage cannot be used as a metric for assessing scalability, performance differences between contexts nor configuration changes: the variance of all cases fit inside the error margin of 5%. The variance of all the values changes between 0,02 percent the minimum being 2,17 and maximum being 2,19%.

There was a possibility that coverage gained would have been different between runs and contexts. In this case there must have been some path constraint that the AFL could not solve because all the instances stalled out in the same point. On the other hand, that given enough time the randomized algorithm in AFL would provide the same kind of results. It is to note however, that although the coverage is virtually the same within all instances, they did find different bugs as detailed in Chapter 4.1.2.

4.2 Unique bug count

As explained in chapter 3.2.2 the unique bug count is derived from the AFL crashing inputs. These inputs are stored and can be examined later. This examination is done by using Crashwalk to deduplicate the crashes via stack hashing. Figure 10 below is an example of Crashwalk outputted summary text file. Crashwalk summary file is different than the run file as it sums up the crashes, but the text file from the run displays every crash its own summary (i.e. one by one basis).

In figure 10 below is displayed the outputted Hash (i.e. *stack hash*), filename for that crash (in this figure 3 files have the same hash, so only the first one is displayed), faulting frame which in this study is used as the name of the crash on chapter 4.1.4 and its GDB output with last three stack entries that the input path took and register values when crash occurred. Extra data in Figure 10 below is an estimation from Crashwalk incorporated tool EXPLOITABLE is not used in this study as it is not quantifiable and rather descriptive in nature.



```
(1 of 3) - Hash: ece62020eef3f0c990bc34f1a0b0183d.ece62020eef3f0c990bc34f1a0b0183d
---CRASH SUMMARY---
Filename: fuzzed/Data/case_11r1/sync_dir/Slave1/crashes/id:000001,sig:11,src:000263+000138,op:splice,rep:32
SHA1: 3435815296bcd3daeb8bcd4488a40e76d07c77d2
Classification: UNKNOWN
Hash: ece62020eef3f0c990bc34f1a0b0183d.ece62020eef3f0c990bc34f1a0b0183d
Command: who fuzzed/Data/case_11r1/sync_dir/Slave1/crashes/id:000001,sig:11,src:000263+000138,op:splice,rep:32
Faulting Frame:
   read_utmp @ 0x000055555558fd76: in /usr/local/bin/who
Disassembly:
   0x000055555558fd62: shr r8,1
   0x000055555558fd65: lea rsi,[r9+r8*1+0x1]
   0x000055555558fd6a: mov QWORD PTR [r15],rsi
   0x000055555558fd6d: imul rsi,r12
   0x000055555558fd71: call 0x55555558a540 <xrealloc>
=> 0x000055555558fd76: mov esi,DWORD PTR [r13+0x16c]
   0x000055555558fd7d: mov edi,0xece
   0x000055555558fd82: mov r12,rax
   0x000055555558fd85: call 0x555555576350 <lava_set>
   0x000055555558fd8a: mov esi,DWORD PTR [r13+0x16c]
Stack Head (3 entries):
   read_utmp                 @ 0x000055555558fd76: in /usr/local/bin/who
   who                       @ 0x0000555555570cca: in /usr/local/bin/who
   main                      @ 0x000055555556df0: in /usr/local/bin/who
Registers:
rax=0x00007ffff79a1010 rbx=0x0000555555b8b490 rcx=0x00007ffff7b063fa rdx=0x0000000000043000
rsi=0x000000000002d000 rdi=0x00007ffff79a1000 rbp=0x0000000000000000 rsp=0x00007fffffffe340
 r8=0x000000000002d000  r9=0x00000000000001d8 r10=0x0000000000000001 r11=0x0000000000000246
r12=0x0000000000000180 r13=0x00007ffff7e26010 r14=0x0000000000000000 r15=0x00007fffffffe360
rip=0x000055555558fd76 efl=0x0000000000010202  cs=0x0000000000000033  ss=0x000000000000002b
 ds=0x0000000000000000  es=0x0000000000000000  fs=0x0000000000000000  gs=0x0000000000000000
Extra Data:
   Description: Access violation on source operand
   Short description: SourceAv (19/22)
   Explanation: The target crashed on an access violation at an address matching the source operand of the current
instruction. This likely indicates a read access violation.
---END SUMMARY---
```

Figure 10 Example of Crashwalk outputted summary text file.

From these text files several unique bugs were uncovered as detailed in table 10 below. In this table I have used the faulting frame as the name of the bug and added its stack has to next column. Next column denotes how many times the said bug was discovered in total through all the runs divided in native and virtual context of multiple case study.

Table 10 Crashwalk deduplicated crashes

	Hash	N	V
read_utmp	ece62020eef3f0c990bc34f1a0b0183d.ece62020eef3f0c990bc34f1a0b0183d	All	All
close_stream	d1b325c43ddba8adf6d50ef78bd394d1.87d90d4b7494464dd0a6b504436ebdfd	3	1
rpl_fclose	9df5d3cc746d835a038b7540382afcd3.7d4fc484b285d5d949e949681b0184f3	1	6
print_user	2ba18b97102e3b0c8ea0b22ec38098e4.2ba18b97102e3b0c8ea0b22ec38098e4	3	0
print_line (1)	4c422aa0ef703a24cad0f81172f78fcf.e09bdbcc905e2b1226e5e941e686be3f	1	3
print_line (2)	566f1a7a8d2bbbfc48cb3ab574d149c8.121589cacfb979a4ec7ba1d71635d1c5	1	0
print_line (3)	4c422aa0ef703a24cad0f81172f78fcf.fa317662eaec46cbc837bd26726cd0c0	0	2
time_string	e51035b678ada00cb89d4ad541d91a65.835aee4c501f6dbe9822d92e8c46fb40	0	1

From table 10 below we can see that read_utmp was found by all instances and all other crashes are very sparsely distributed between all instances. Further data analysis is done in chapter 4.1.4 where analysis focuses on using these bugs as metric for performance analysis.

Below on tables 11-14 I have included the instance specific information on how many crashes AFL (SUM AFL) has reported and how many Crashwalk (SUM CW) has deduplicated total crashes for the run on specific instance. Minimum and maximum values are included as additional information in order to deduce what configuration discovered most crashes (RQ3). Average is counted from deduplicated bugs (CW avg.) and marked up to decipher the general crash discovering capability of a run of an instance.

Table 11 Case 2 crash count and deduplicated bugs

	Native environment					Virtual Environment				
1m1s	**r1**	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
SUM AFL	3	3	3	3	3	4	2	3	5	4
SUM CW	1	1	1	1	2	2	1	1	2	1
MIN CW	0	0	0	0	1	1	1	1	1	1
MAX CW	1	1	1	1	1	1	1	1	1	1
Best Instc.	S1	S1	S1	S1	S1	All	All	All	All	All
CW avg.	0,5	0,5	0,5	0,5	1	1	1	1	1	1
2m	**r1**	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
SUM AFL	7	3	3	1	1	5	6	6	4	6
SUM CW	2	2	2	1	1	2	2	2	2	2
MIN CW	1	1	1	0	0	1	1	1	1	1
MAX CW	1	1	1	1	1	1	1	1	1	1
Best Instc.	All	All	All	M2	M2	All	All	All	All	All
CW avg.	1	1	1	0,5	0,5	1	1	1	1	1

From table 11 is shown that AFL has overcounted crashes by 150% to 400%. On native execution only 4 runs out of 10 have found a second bug whereas on virtual context 7 out of 10 have found a second bug. In Native environment slaves were better at finding crashes in 5 out of 5 cases, as the 2m instance only holds master instances. On virtual context both masters and slaves found crashes evenly in all runs.

Table 12 Case 3 crash count and deduplicated bugs

	Native environment					Virtual Environment				
1m2s	**r1**	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
SUM AFL	7	6	6	6	8	7	8	6	6	6
SUM CW	3	2	2	2	3	2	3	2	2	2
MIN CW	0	0	0	0	1	0	0	0	0	0
MAX CW	2	1	1	1	1	1	2	1	1	1
Best Instc.	S2	S1 -2	S1-2	S1	All	S1-2	S2	S1-2	S1-2	S1-2
CW avg.	1	0,67	0,67	0,67	1	0,67	1	0,67	0,67	0,67

2m1s	r1	r2	r3	r4	r5	r1	r2	r3	r4	r5
SUM AFL	3	2	8	6	2	4	5	6	4	7
SUM CW	2	1	3	3	1	2	3	4	2	3
MIN CW	0	0	1	1	0	0	1	1	0	1
MAX CW	1	1	1	1	1	1	1	2	1	2
Best Instc.	M1 S1	S1	All	All	S1	M1 S1	All	S1	M2 S1	S1
CW avg.	0,67	0,33	1	1	0,33	0,67	1	1,33	0,67	1,5
3m	**r1**	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
SUM AFL	7	9	8	2	9	10	8	8	8	6
SUM CW	3	3	4	1	3	3	3	3	3	2
MIN CW	1	1	1	0	1	1	1	1	1	1
MAX CW	1	1	2	1	1	1	1	1	1	1
Best Instc.	All	All	M3	M1	All	All	All	All	All	All
CW avg.	1	1	1,33	0,33	1	1	1	1	1	1

On table 12 AFL has overcounted crashes by 166% to 300%. On native execution 2 runs out of 15 have found a second crash whereas on virtual context 3 out of 15 have found a second crash. Native environment slaves were better at finding crashes in 6 out of 10 cases, as the 3m instance only holds master instances. This trend continues in Virtual context as 7 out of 10 instances report slaves finding more or the only crash of instance.

Table 13 Case 4 crash count and deduplicated bugs

	Native environment					Virtual Environment				
1m3s	**r1**	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
SUM AFL	12	9	9	12	9	11	9	7	10	10
SUM CW	4	4	4	3	3	3	3	3	3	3
MIN CW	1	1	1	0	0	0	0	0	0	0
MAX CW	1	1	1	1	1	1	1	1	1	1
Best Instc.	All	All	All	S1-3	S1-3	S1-2	S1-2	S1-2	S1-2	S1-2
CW avg.	1	1	1	0,75	0,75	0,75	0,75	0,75	0,75	0,75
2m2s	**r1**	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
SUM AFL	11	7	9	8	10	23	11	13	6	12
SUM CW	4	3	3	3	3	3	4	4	3	4
MIN CW	1	0	0	0	0	0	1	1	0	1
MAX CW	1	1	1	1	1	1	1	1	1	1
Best Instc.	All	M2 S1-2	M2 S1-2	M1 S1-2	M2 S1-2	M1 S1-2	All	All	M1 S1-2	All
CW avg.	1	0,75	0,75	0,75	0,75	0,75	1	1	0,75	1

3m1s	r1	r2	r3	r4	r5	r1	r2	r3	r4	r5
SUM AFL	10	9	11	10	10	6	7	11		9
SUM CW	4	4	4	4	4	3	4	4		3
MIN CW	1	1	1	1	1	0	1	1		0
MAX CW	1	1	1	1	1	1	2	1		1
Best Instc.	All	All	All	All	All	M1,3 S1	S1	All		M2-3 S1
CW avg.	1	1	1	1	1	0,75	1,333333	1		0,75

AFL has overcounted crashes by 200% to 766% as shown in table 13 above. On native execution none out of 15 have found a second crash whereas on virtual context 1 instance found a second crash. Native environment slaves were better at finding crashes in 2 out of 15 cases. On Virtual context 6 out of 14 instances report slaves finding more or the only crash of instance.

Crash finding ability seems to be leveling out on Case 4 as the masters start to find crashes during the 24 hour period and in Native context 9 instances out of 15 (60%) report all instances finding a crash, whereas in Virtual context 4 out 14 (~29%) has all instances finding a crash. Comparing results to Case 3 (and its table 12 far above) where 3 out of 10 (30%) Native context mixed instances (masters and slaves present) and 1 out of 10 (10%) Virtual instances have all instances finding a crash.

If we take into account the homogenous master instances of Case 3 the percentages change: 6/15 (40%) Case 3 native and virtual whereas 9/15 (60%) for Case 4 Native and 4/15(~29%) on Virtual context. Which still supports the claim of the ability to find crashes starts to even out in this point between configuration changes.

Table 14 Case 5 crash count and deduplicated bugs

	Native environment					Virtual Environment				
1m7s	r1	r2	r3	r4	r5	r1	r2	r3	r4	r5
SUM AFL	21	27	27	27	29	25	26	24	23	22
SUM CW	7	9	8	8	7	10	8	9	7	7
MIN CW	0	1	1	1	0	0	1	1	0	0
MAX CW	1	2	1	1	1	3	1	2	1	1
Best Instc.	S1-7	S5	All	All	All	S7	All	S7	S1-7	S1-7
CW avg.	0,875	1,125	1	1	0,875	1,25	1	1,125	0,875	0,875
2m6s	**r1**	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
SUM AFL	21	28	24	20	20	20	18	15	21	24
SUM CW	8	8	8	6	7	7	6	7	10	8
MIN CW	1	1	1	0	0	1	1	0	1	1
MAX CW	1	1	1	1	2	2	1	1	2	1
Best Instc.	All	All	All	S1-6	S2	S4-6	All	M1 S1-6	S2,4	All

CW avg.	1	1	1	0,75	0,875	1,375	1	0,875	1,25	1
3m5s	**r1**	**r2**	**r3**	**r4**	**r5**	**r1**	**r2**	**r3**	**r4**	**r5**
SUM AFL	22	20	21	21	24	20	18	15	21	24
SUM CW	7	7	8	7	9	7	6	7	10	8
MIN CW	0	0	1	0	1	1	1	1	1	1
MAX CW	1	1	1	1	2	1	1	1	3	1
Best Instc.	M1,3 S1-5	M2-3 S1-5	All	M1,3 S1-5	S2	All	All	All	S3	All
CW avg.	0,875	0,875	1	0,875	1,125	1	1	1	1,25	1

From table 14 above we can deduce that AFL has overcounted crashes by ~214% to ~338%. On native execution 3 out of 15 have found a second crash whereas on virtual context 5 instance found a second crash and 2 out of 15 a third one. Native environment slaves were better at finding crashes in 5 out of 15 cases. On Virtual context 7 out of 14 instances report slaves finding more or the only crash of instance.

From Case 5 Native and Virtual context 7 out of 15 (~47%) instances were able to tie on finding a crash. In previous Case 4 Native context tied on 40% of instances and Virtual context 60%. From this data we cannot deduce if configuration changes affect the amount of crashes found by a run of AFL.

Between all cases there are only 11 runs out of 110 (10%) that have found a second (or third in V 3m 5s r4) crash. From the remaining 99 runs all have found at least one crash. From this data it is impossible to decipher statistical differences as most of the data is homogenous. Therefore, this metric of counting the number of crashes found by an run with different configurations is not usable for performance testing in this study as it did not have enough dispersion.

4.3 Execution speed

For execution speed metric the minimum and maximum values together with standard deviation are reported on Appendix 2, as they do not directly contribute to performance metric, but are essential in assessing the validity of this study. In the Chapter 4.4.1 tables 15-22 the execution speed is averaged per instance and run. From the data a weighted average execution speed (WAES) is counted per run (marked darker in the tables), as the runs have a vast difference in datapoints: slave instances report data far more frequently than master instances in all cases.

Xu et al. have used execution speed as a metric in their study as described in chapter 3.2.4. Assessing from the data in the study the worker execution speed is averaged over the runtime of fuzzer. This also means that the execution speed is cumulated over worker (Xu et al. ″core″), meaning that in order to have comparable results a cumulative average execution speed (CAES) must be counted and represented. CAES of all the runs is displayed in Table 23 (Chapter 4.4.2). These values along with WAES are displayed in Figures 11-14 on Chapter 4.4.2.

4.3.1 Execution speed (Weighted Averages)

Table 15 Case 2 Native instance average speeds and Weighted average.

	r1	r2	r3	r4	r5
	Datapoints / Avg.	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**
1m1s	1197,83	1092,36	1079,71	1277,86	1211,50
M1	143 /1342,09	110/1276,19	114/1346,61	220/1382,83	107/1353,86
S1	16096/1196,54	14953/1090,99	15616/1077,75	16354/1276,44	15691/1210,52
2m	1480,96	1322,35	1368,84	1326,36	1431,50
M1	1386/1506,28	272/1300,62	312/1371,51	411/1370,88	995/1443,12
M2	1029/1446,87	1087/1327,81	1276/1368,18	895/1305,89	531/1409,73

Table 15 shows the difference in weighted averages of instances. From datapoint count we can see that master instances do not record as diligently as slave instances, and even in two-master instance they do not record at the same pace. From weighted averages we can deduce that the two-master instance has higher average execution speed in all cases.

Table 16 below shows the difference in weighted averages of instances. Same kind of datapoint count trend persists in virtual execution as in native execution. From weighted averages we can deduce that the two-master instance has higher average execution speed in all cases.

Table 16 Case 2 Virtual instance average speeds and Weighted average.

	r1	r2	r3	r4	r5
	Datapoints / Avg.	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**
1m1s	1247,00	1123,12	1290,27	1114,10	1318,65
M1	138/1577,99	97/1557,27	188/1509,63	128/1496,85	133/1570,92
S1	15988/1244,13	16210/1120,50	16471/1287,76	15751/1110,97	15806/1316,51
2m	1444,13	1553,47	1587,96	1481,63	1534,24
M1	1152/1468,57	5547/1555,36	1815/1569,38	1377/1477,53	454/1511,96
M2	384/1370,93	345/1523,14	1425/1611,63	1457/1485,50	1746/1540,05

Table 17 below shows that there is difference in weighted averages of instances. Datapoint trend seems to emerge here as slave instances seem to record diligently and master instances somewhat sparsely. Weighted averages between runs and instances start to mix but favor the three-master instance in all cases.

Table 17 Case 3 Native instance average speeds and Weighted average.

	r1	r2	r3	r4	r5
	Datapoints / Avg.	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**
1m2s	1218,67	1259,43	1162,44	1166,92	1082,78
M1	128/1353,74	151/1476,50	124/1293,54	186/1409,78	78/1336,60
S1	16864/1190,89	16901/1253,27	16697/1154,22	16585/1159,63	16819/1090,56
S2	16887/1245,38	16615/1263,70	16905/1169,59	16859/1171,40	16745/1073,78
2m1s	1196,99	1212,29	1355,99	1013,88	1189,73
M1	130/1276,28	295/1290,66	7934/1404,42	160/1240,69	232/1314,22
M2	155/1042,24	250/1282,07	256/1377,12	132/1251,71	222/1264,00
S1	16509/1197,83	16051/1209,75	16609/1332,53	15528/1009,49	15602/1186,81
3m	1225,85	1354,69	1374,77	1262,70	1425,26
M1	1196/1219,93	3725/1318,73	3360/1353,90	4627/1244,44	2089/1439,07
M2	2772/1219,44	3541/1385,57	2007/1408,89	267/1267,96	715/1460,22
M3	429/1283,72	5549/1359,13	3643/1375,22	785/1368,43	2161/1400,33

Table 18 below shows difference in weighted averages of instances. Datapoint trend set in previous paragraph persists, although it seems that in most cases native context masters have recorded more datapoints when compared pair wisely (17 wins for native, 13 for virtual). Weighted averages between runs and instances start to mix but favor the three-master instance in all cases.

Table 18 Case 3 Virtual instance average speeds and Weighted average.

	r1	r2	r3	r4	r5
	Datapoints / Avg.	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**
1m2s	1066,99	1345,81	1346,65	1234,71	1266,49
M1	122/1453,32	207/1474,69	190/1495,26	147/1558,67	126/1515,16
S1	16714/1077,37	16905/1343,30	16914/1360,95	16852/1258,97	16574/1272,76
S2	16845/1053,88	16867/1346,73	16905/1330,66	16711/1207,38	16918/1258,49
2m1s	1151,97	1340,72	1282,35	1296,10	1134,50
M1	276/1351,00	170/1337,51	176/1389,10	266/1367,27	227/1352,16
M2	233/1279,26	178/1420,00	149/1374,19	221/1349,75	109/1434,64
S1	16606/1146,85	16694/1339,90	16506/1280,37	16524/1294,23	16506/1129,50
3m	1374,91	1335,67	1361,06	1383,30	1258,84
M1	5599/1362,17	1634/1287,84	837/1415,03	1758/1355,92	3612/1252,23
M2	1558/1468,85	494/1403,93	437/1315,18	1611/1405,16	2917/1267,06
M3	2365/1343,18	319/1474,43	241/1257,24	1406/1392,48	344/1258,63

Table 19 Case 4 Native instance average speeds and Weighted average.

	r1	**r2**	**r3**	**r4**	**r5**
	Datapoints / Avg.	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**
1m 3s	851,56	1149,71	931,32	1234,26	1082,65
M1	158/1385,92	102/1399,96	66/1302,47	165/1334,47	119/1326,70
S1	16327/618,97	16841/1146,26	16783/940,82	16956/1232,92	16909/1167,42
S2	16931/1303,21	16887/1148,75	16744/933,80	16960/1235,80	16749/1032,26
S3	16814/617,57	16883/1152,58	16747/917,85	16948/1233,08	16845/1045,91
2m 2s	1029,87	1158,43	931,47	1160,85	1238,40
M1	129/1189,50	174/1204,34	116/1198,60	177/1187,14	134/1327,65
M2	139/1263,07	137/1219,83	176/1222,38	134/1263,94	169/1286,11
S1	16778/1040,89	16923/1174,03	16626/791,86	16894/1110,66	16940/1253,95
S2	16631/1015,55	16910/1141,84	16805/1064,66	16822/1210,14	16920/1221,63
3m 1s	1011,26	1198,16	1204,12	1222,73	1211,40
M1	277/1103,86	219/1272,34	277/1301,35	765/1234,21	1130/1227,67
M2	174/1197,78	157/1222,91	1286/1167,89	235/1213,32	250/1136,09
M3	203/1100,43	1877/1145,08	313/1359,62	2172/1227,46	2462/1250,13
S1	16121/1006,51	16506/1202,98	16222/1202,31	16684/1221,72	16050/1205,48

Difference in weighted averages of instances continues in Table 19 above. Datapoint trend set in previous paragraphs persists. Weighted averages seem to drop somewhat compared to Case 3 (tables 17 and 18) but still mix together: 1m3s has one "win" (i.e. best WEAS in r4), 2m2s also has one win (r5) and 3m1s instance wins rest three instances.

This average speed leveling out is most probably due to the fact that slave instances record far more datapoints with lower average scores than master instances. This skews the average towards slave instance execution speed. This is apparent in all 3m1s runs (on Table 19) where the slave has magnitudes more datapoints than all combined masters while having equal level or worse execution speed.

Table 20 Case 4 Virtual instance average speeds and Weighted average.

	r1	r2	r3	r4	r5
	Datapoints / Avg.	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**
1m 3s	1245,17	1098,33	1195,04	946,80	1162,91
M1	139/1462,74	169/1502,80	137/1500,11	78/1405,46	118/1622,00
S1	16952/1239,00	16853/1095,13	16920/1186,74	16815/946,23	16867/1158,75
S2	16922/1251,78	16859/1115,25	16892/1208,05	16846/948,77	16823/1160,54
S3	16947/1242,94	16872/1080,55	16928/1187,87	16824/943,24	16883/1166,19
2m 2s	1142,08	1155,44	1050,69	1339,70	1333,02
M1	152/1383,11	227/1408,62	164/1338,89	220/1360,20	308/1305,87
M2	113/1329,45	198/1392,55	139/1290,95	181/1396,06	183/1400,76
S1	16258/1142,97	16842/1167,78	16476/1060,60	16920/1336,49	16778/1350,93
S2	16836/1137,76	16875/1136,91	16525/1035,90	16806/1342,04	16946/1315,06
3m 1s	1141,95	1199,74	1290,76		1117,78
M1	298/1415,34	256/1326,10	1479/1336,89		192/1265,16
M2	222/1328,86	275/1350,78	1199/1331,22		163/1372,42
M3	30371367,98	206/1276,36	250/1287,90		152/1270,12
S1	16456/1130,27	16691/1194,34	16522/1283,73		15870/1111,89

Table 20 shows that the there is difference in weighted averages of instances. Datapoint trend set in previous paragraphs persists, and most probably skews the WEAS in all instances towards slave instance execution speed. Weighted averages seem to drop somewhat compared to Case 3 (Tables 17 and 18) but do not exhibit sub 1000 execution as native context does (Table 19 above).

Table 21 Case 5 Native instance average speeds and Weighted average.

	r1	r2	r3	r4	r5
	Datapoints / Avg.	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**
1m 7s	1173,23	1101,57	979,26	1043,15	955,51
M1	134/1261,31	140/1306,79	72/1254,53	122/1369,49	111/1355,70
S1	16980/1217,18	16932/1105,90	16896/986,16	16899/1107,37	16897/1085,13
S2	16979/1211,18	16934/1090,52	16859/992,56	16902/1105,96	16792/540,73
S3	16980/1221,58	16926/1116,75	16896/980,24	16911/1066,64	16900/1155,68
S4	16966/1210,05	16937/1093,13	16840/947,99	16907/1073,69	16801/540,80
S5	16976/1213,94	16935/1101,16	16903/945,91	16906/1081,05	16928/1158,52
S6	16983/1219,43	16916/1099,00	16838/1020,64	16909/1073,91	16926/1070,04
S7	16939/917,94	16933/1102,81	16905/980,24	16837/790,08	16917/1129,30
2m 6s	1069,99	963,95	998,03	1052,19	1014,38
M1	185/1223,72	128/1286,59	138/1154,29	159/1191,86	164/1307,45
M2	102/1223,41	163/1203,78	139/1209,82	160/1176,60	157/1266,46
S1	16918/1047,48	16893/993,11	16846/993,17	16843/806,63	16918/1214,49
S2	16948/1091,14	16793/719,13	16842/1010,32	16914/1091,04	16962/1215,57
S3	16926/1051,62	16896/1027,03	16889/1008,27	16908/1085,81	16864/589,61
S4	16934/1049,01	16823/992,33	16847/1001,55	16895/1103,72	16957/1247,28
S5	16943/1099,37	16879/1040,93	16865/980,49	16913/1110,50	16854/589,59
S6	16949/1078,62	16875/1005,16	16890/991,38	16911/1111,98	16955/1220,08
3m 5s	1260,55	1055,36	997,59	1213,18	1246,54
M1	3262/1244,45	212/584,41	156/1130,61	3965/1250,08	1010/1200,42
M2	236/1240,46	328/1225,50	186/1125,58	127/1244,65	183/1229,48
M3	6467/1266,50	245/1218,16	246/1092,41	6741/1254,48	255/1198,00
S1	16977/1252,57	16953/1215,72	16874/785,36	16966/1215,81	16945/1246,25
S2	16980/1244,37	16935/1242,48	16948/1066,47	16963/1230,47	16915/1233,84
S3	16977/1286,51	15947/367,83	16950/1042,85	16957/1207,22	16924/1264,24
S4	16976/1264,88	16953/1195,64	16924/1032,38	16970/1206,80	16949/1245,35
S5	16974/1255,52	16941/1214,90	16952/1055,99	16968/1180,31	16955/1246,71

Table 22 Case 5 Virtual instance average speeds and Weighted average.

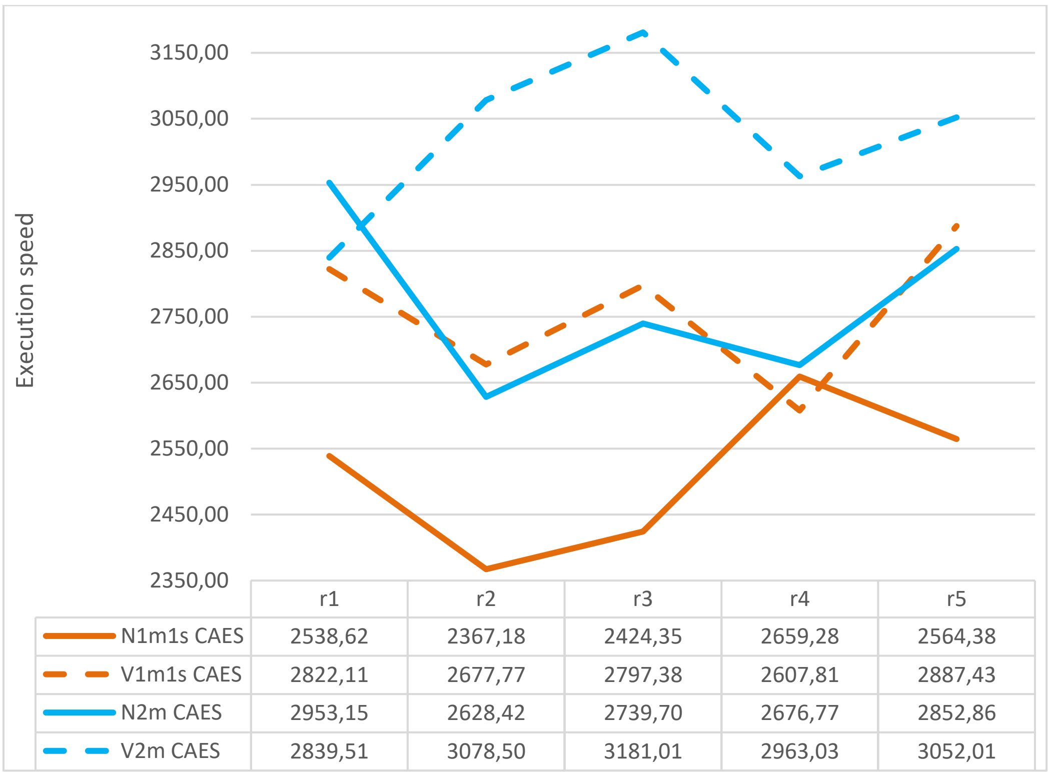
	r1	r2	r3	r4	r5
	Datapoints / Avg.	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**	**Datapoints / Avg.**
1m 7s	1262,53	1022,89	1072,16	1094,23	1114,53
M1	143/1374,78	78/1397,63	86/1470,06	98/1427,50	101/1443,97
S1	16972/1261,42	16855/1015,67	16885/1075,59	16919/1070,53	16943/1103,13
S2	16971/1261,49	16860/1029,86	16860/1070,99	16905/1083,02	16909/1127,37
S3	16976/1260,61	16703/1021,15	16886/1075,35	16921/1094,91	16941/1126,70
S4	16976/1265,98	16855/998,51	16910/1064,71	16902/1104,20	16952/1104,24
S5	16975/1269,59	16858/1023,03	16901/1077,66	16912/1105,59	16949/1119,56
S6	16977/1258,80	16852/1047,40	16908/1063,90	16918/1098,64	16941/1105,34
S7	16970/1258,90	16862/1022,87	16911/1074,86	16912/1100,76	16946/1113,42
2m 6s	975,57	960,99	1036,36	1187,79	972,49
M1	190/1385,46	112/1338,97	118/1279,85	213/1275,23	142/1200,38
M2	183/1345,88	95/1399,87	66/665,64	170/1334,80	119/1225,38
S1	16822/564,11	16830/975,24	16889/1211,12	16953/1208,73	16868/964,23
S2	16920/1173,75	16859/954,07	16916/1170,99	16954/1200,86	16869/962,33
S3	16930/1169,61	16824/971,61	16905/1143,85	16940/1171,71	16811/992,20
S4	16916/1181,22	16835/950,45	16262/313,99	16948/1215,49	16858/974,94
S5	16828/563,69	16853/948,81	16913/1188,76	16967/1175,64	16881/956,37
S6	16919/1187,70	16874/960,72	16914/1161,67	16892/1151,62	16876/981,18
3m 5s	1275,54	1140,78	1106,13	1151,52	954,84
M1	252/1300,47	109/1248,36	183/1246,93	178/508,15	171/1247,86
M2	217/1297,73	148/1216,79	210/1196,28	155/614,25	128/1293,77
M3	5820/1267,62	273/1312,46	122/1303,45	216/1257,70	135/1264,85
S1	16931/1259,46	16852/1130,99	16901/1114,90	16909/1159,08	16853/947,45
S2	16946/1260,40	16915/1139,41	16895/1124,66	16889/1169,66	16867/954,38
S3	16946/1276,75	16913/1144,43	16877/1092,18	16861/1142,13	16884/944,46
S4	16946/1278,15	16882/1137,19	16887/1096,83	16909/1153,69	16886/957,06
S5	16944/1305,00	16898/1147,69	16892/1097,99	16903/1143,45	16844/962,79

Table 21 and 22 show that there is difference in weighted averages of instances and context. From Table 21 and 22 it is clear, that slave instance datapoints and averages skew the weighted average towards their own averages. It seems both 1mx and 2mx cases the execution speed has dropped in both execution environments but is marginally the same with 3mx instance.

Essentially Weighted Averages of Execution speed skew towards datasets created by Slave instances. From this we see a trend that contradicts Xu et. al (Xu, Kashyap, Min, & Kim, 2017) results of execution speed increasing as more cores are added (until 15 cores are being used). Weighted average was supposed to scale up as cores were added, because the added instances could have been doing more work and distributing the test case generators que more effective. This might still be the case internally, but the metric in this case only considers the slave instance leveraged average execution speed. As this metric contradicts common knowledge and is tainted by slave datapoints it is not used as scalability metric. Therefore, Xu et. al have used cumulative execution speed as a metric in their study.

4.3.2 Execution speed (Cumulative Average execution speed))

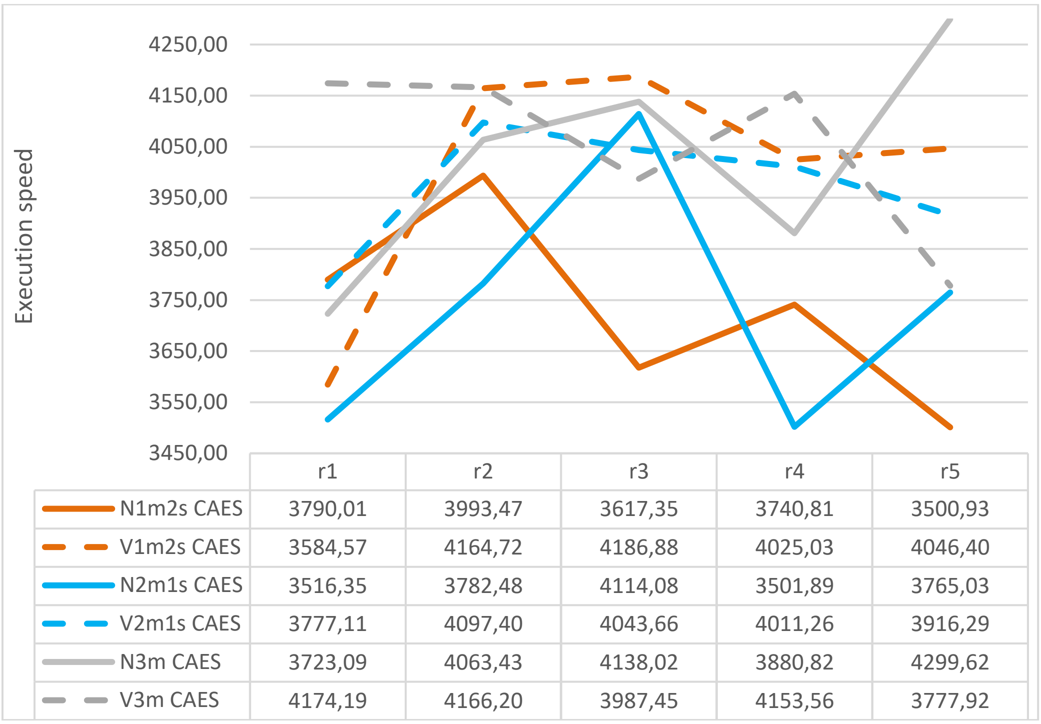
Cumulative execution speed average (CAES) is counted by summing up the instance averages. This is done in order not to skew the average values between master and slave instance due to datapoint imbalance showed in previous chapter 4.1.3. CAES values are visualized in Figures 11-14 below and their values are derived from Tables 15 -22. These figures are displayed in case-by-case basis with on both contexts. Naturally in execution speed higher values are better.



	r1	r2	r3	r4	r5
N1m1s CAES	2538,62	2367,18	2424,35	2659,28	2564,38
V1m1s CAES	2822,11	2677,77	2797,38	2607,81	2887,43
N2m CAES	2953,15	2628,42	2739,70	2676,77	2852,86
V2m CAES	2839,51	3078,50	3181,01	2963,03	3052,01

Figure 11 Case 2 CAES visualized

In Figure 11 above we see the CAES of case 2 in both contexts, native on continuous- and virtual as dashed line. One-master instances colored orange and two-master instances blue. It is apparent that in 4 out of 5 cases the CAES of virtual instance is higher than its native counterpart in two core instances. Furthermore, two-master instance is faster in its own context, although they are close for example in r4 of native context.

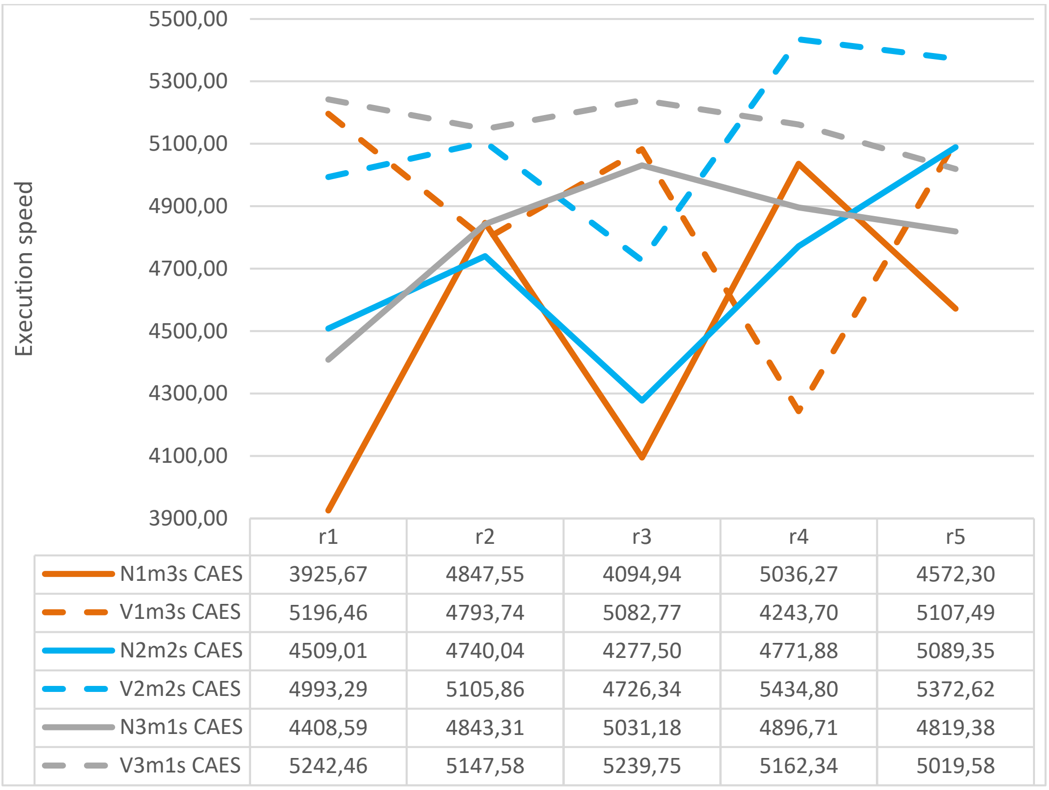


	r1	r2	r3	r4	r5
N1m2s CAES	3790,01	3993,47	3617,35	3740,81	3500,93
V1m2s CAES	3584,57	4164,72	4186,88	4025,03	4046,40
N2m1s CAES	3516,35	3782,48	4114,08	3501,89	3765,03
V2m1s CAES	3777,11	4097,40	4043,66	4011,26	3916,29
N3m CAES	3723,09	4063,43	4138,02	3880,82	4299,62
V3m CAES	4174,19	4166,20	3987,45	4153,56	3777,92

Figure 12 Case 3 CAES visualized

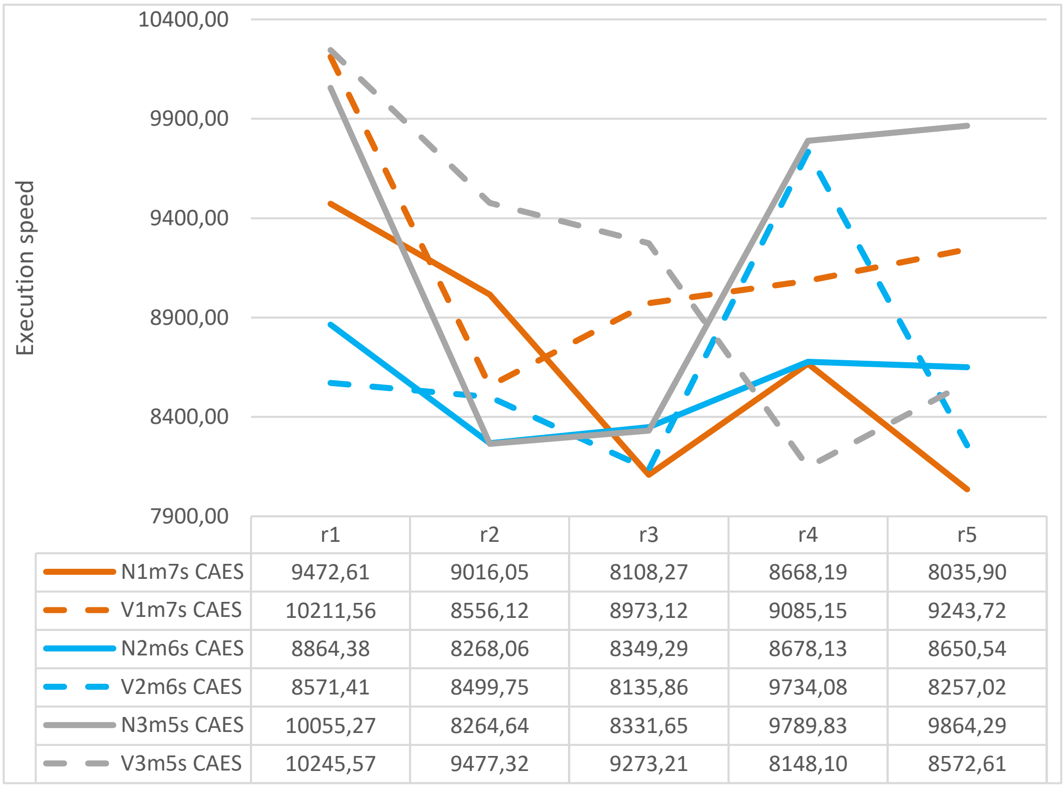
In Figure 12 above we see the CAES of case 3 in both contexts, using same line graphics as in previous figure of Case 2. One-master instances colored orange, two-master instances blue and three-master instance grey. In one- and two-master instance same trend of 4 out of 5 CAES being higher than native context. For three-master instance the results are a bit more mixed, virtual context being better only in 3 out of 5 cases. Generally (4 out of 5) we see three-master instance beating one- and two-master instances in CAES in native context, but evening out on virtual context (3 out of 5).

Figure 13 below follows the same line and coloring scheme as Figure 12 and shows the results of Case 4 CAES. As the datapoint of V3m1sr4 is forever lost it is replaced with the average value of other four runs in order to visualize this comparison. The gap between virtual and native execution starts to widen, as we can see only two runs being better than its virtual counterpart (N1m3s r2 and r4). As per Case 3 we see three-master instance dominating over two- and one-master instances in 4 out of 5 cases in native context, but the role is reversed in virtual context as instance is better in only 3 out of 5 runs.



	r1	r2	r3	r4	r5
N1m3s CAES	3925,67	4847,55	4094,94	5036,27	4572,30
V1m3s CAES	5196,46	4793,74	5082,77	4243,70	5107,49
N2m2s CAES	4509,01	4740,04	4277,50	4771,88	5089,35
V2m2s CAES	4993,29	5105,86	4726,34	5434,80	5372,62
N3m1s CAES	4408,59	4843,31	5031,18	4896,71	4819,38
V3m1s CAES	5242,46	5147,58	5239,75	5162,34	5019,58

Figure 13 Case 4 CAES visualized



	r1	r2	r3	r4	r5
N1m7s CAES	9472,61	9016,05	8108,27	8668,19	8035,90
V1m7s CAES	10211,56	8556,12	8973,12	9085,15	9243,72
N2m6s CAES	8864,38	8268,06	8349,29	8678,13	8650,54
V2m6s CAES	8571,41	8499,75	8135,86	9734,08	8257,02
N3m5s CAES	10055,27	8264,64	8331,65	9789,83	9864,29
V3m5s CAES	10245,57	9477,32	9273,21	8148,10	8572,61

Figure 14 Case 5 CAES visualized

Figure 14 above follows the same coloring and line scheme as previous figures in this chapter. Something interesting is happening at Case 5 because the results start to mix up. Most likely the results start to mix up as the slave instances start to have more impact on cumulative execution speed because they have less deviation than master instances (standard deviations are reported in Appendix 2).However there is a clear difference between native and virtual context: one- and three-master instances virtual counterparts perform better than their native, but the role is reversed in two-master instance.

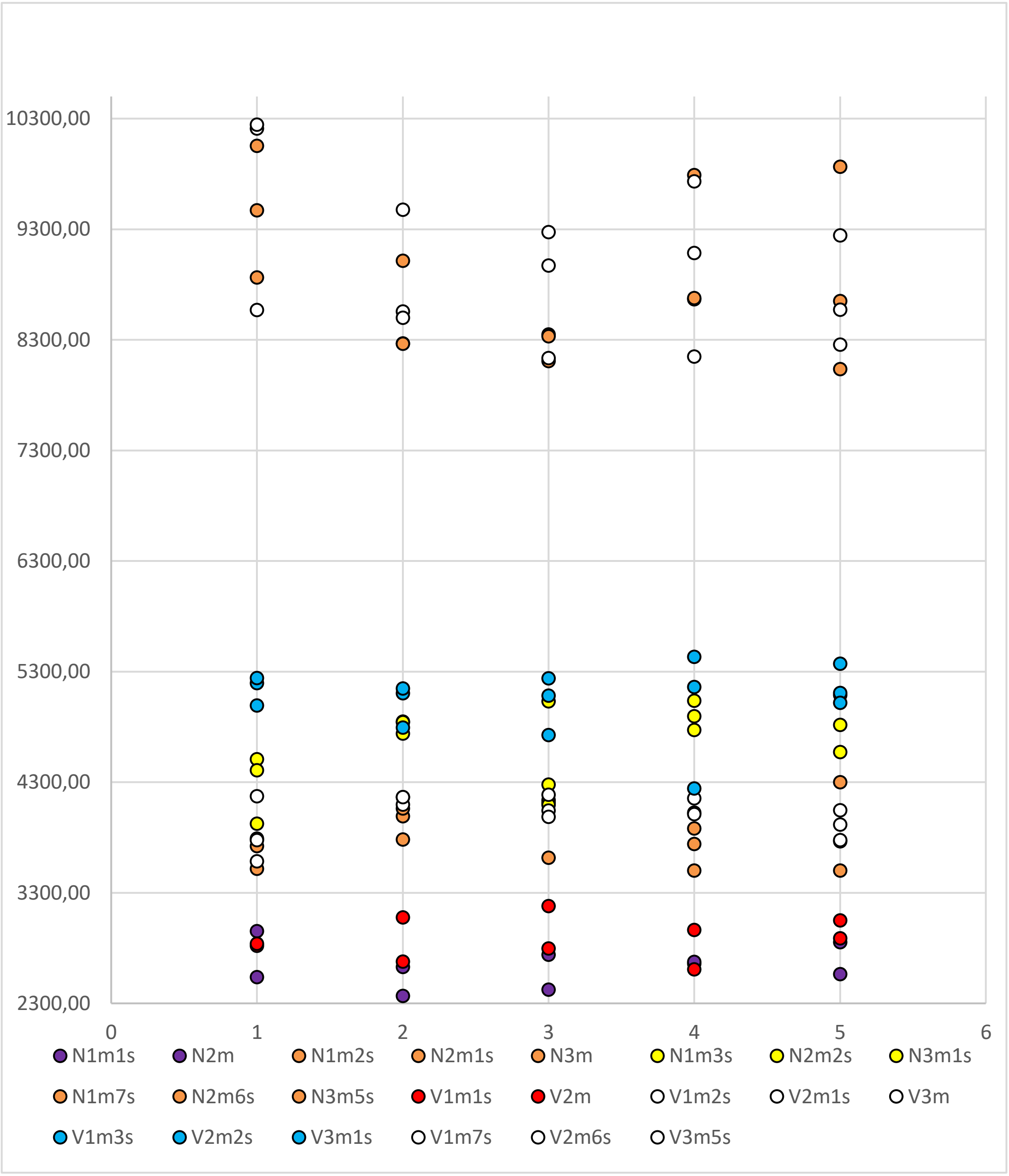


Figure 15 All cases CAES combined

Figure 15 above plots all the cases on the same plot using the run number as X-axis and execution speed as Y-axis. Cases are designed their own color, except cases 3 and 5 which have the same coloring schema but are visually located so far apart that the coloring is irrelevant. From Figure 15 we can deduce that Cases 2 (Violet for native execution, red for virtual) and 5 (native on orange, virtual on white) form their own grouping through all the runs, where Cases 3 and 4 intermingle between 3300 and 5300 execution speed with lest distinction. This plot shows that there is a continual execution speed increase when increasing workers and this presumption is tested in Chapter 5.

Cumulative execution speed as a metric show promising results to calculate if speedup is present. These results are further analyzed in Chapter 5. As far as results and their predictivity go, the one- and three-master instances follow a very predictive path, but two-master instance has large amount of variation in its results. This would require additional investigation on why two-master instance has this much dispersion in its results.

4.4 Shared bugs

Table 23 Found shared bugs in both multiple case contexts

Identified bug	**1m1s**	**2m**	**1m2s**	**2m1s**	**3m**	**1m3s**	**2m2s**	**3m1s**	**1m7s**	**2m6s**	**3m5s**
read_utm	B	B	B	B	B	B	B	B	B	B	B
close_stream			N						N		B
rpl_close			V	V	N			V	V		V
print_user									N		
print_line (1)				V			V		V	N	
print_line (2)									N		
print_line (3)							V			V	
time_string										V	

Table 23 above shows the discovered crashes in all cases and instances. These crashes were verified from Crashwalk *stack hashes* to be unique as stated in chapter 4.3. In this table the instance is denoted as "B" if both virtual and native execution contexts have found the same bug. Bug is denoted N or V for Native and virtual context discoveries respectfully.

From this data we can deduce that all other bugs other than "read_utm" are too sparse to be able to analyze. This is to be expected as fuzzer algorithm is random in nature, so it treads different paths and those paths probably did not converge in this case study. This "read_utm" bug is most likely what you would call a "shallow bug" as it is found by all instances.

There are great differences in discovered bugs between native and virtual context. Native has found 6 unique bugs and virtual context has found 5. Most notably "time_string" bug is the only one found in virtual context and “print_user” found in native context. These could be called “deep bugs” but as the coverage stayed low it cannot (see chapter 4.1.1) be said certainly that they reside deep in the target program.

4.4.1 Shared bug (read_utmp)

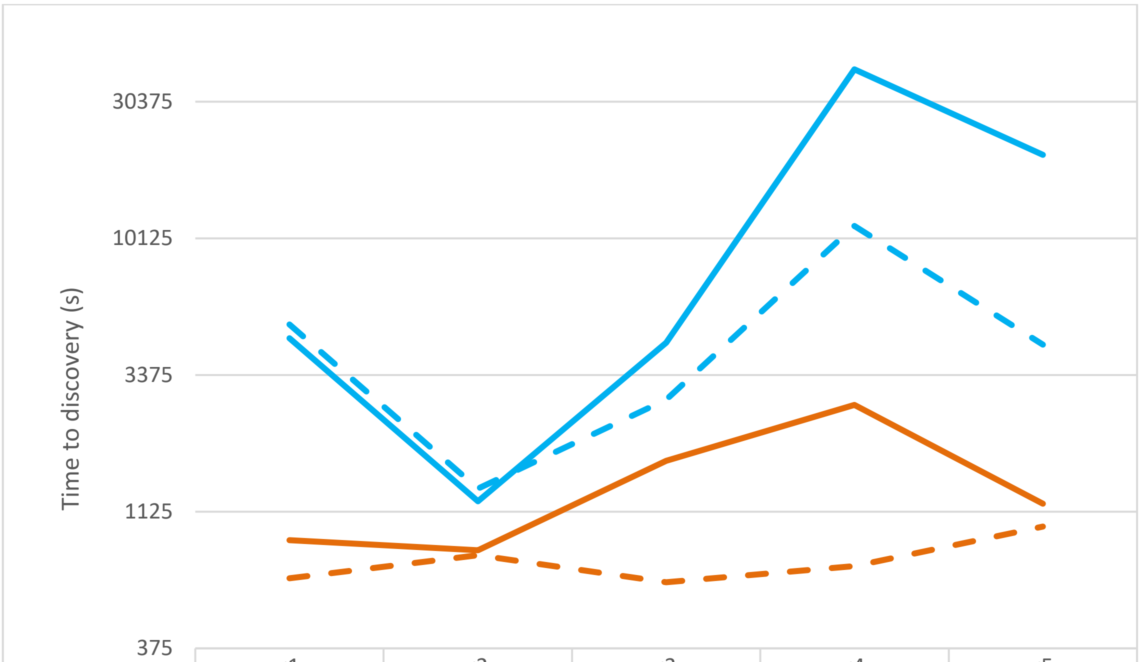
On the following table 24 shows the fastest time a worker instance has found bug “read_utmp” in time elapsed as seconds from starting the instance. This table follows the same naming convention for instances and runs as all the Chapter 4 tables. Both virtual and native environment results are displayed.

Fastest time is derived by searching all the instances that have crashing input to read_utmp (from Crashwalk instance specific files) and then deducing what was the first instance of AFL crashing input-file stored on the AFL directory that the Crashwalk output referred to. AFL starts numbering from id:0000 onward, and therefore, the first unique crash is id:0000 and so forth. Therefore, AFL crashes can be checked against the Crashwalk output and counted how many seconds elapsed since AFL had detected the crash from the AFL instances plot file.

Table 24 First discovery of read_utmp bug from each instance

	Native					Virtual				
	r1	r2	r3	r4	r5	r1	r2	r3	r4	r5
1m1s	895	825	1692	2654	1199	658	792	638	727	998
2m	4534	1223	4376	39397	19837	5068	1354	2767	11186	4310
1m2s	858	919	736	771	889	971	1453	731	902	746
2m1s	1363	27751	753	696	695	1333	725	918	675	909
3m	7147	800	10847	74577	24340	7343	882	5211	1712	27509
1m3s	629	654	736	962	789	859	811	772	967	1230
2m2s	960	22118	1493	4230	1219	829	704	747	686	917
3m1s	645	3251	706	1032	741	669	1780	1515		778
1m7s	665	687	669	726	669	810	681	870	684	629
2m6s	802	1384	655	1204	958	660	627	674	636	856
3m5s	667	657	728	778	670	647	664	996	628	734

Table 24 above is presented per case basis in following figures 16-19 where one-master instances are colored orange, two-master instances blue and three-master instances grey. Continuous line depicts the native context and dashed line the virtual context, as was the case in Chapter 4.4.2. The difference to Chapter 4.4.2 figures is that lower discovery speed is better, as bugs are best found young.

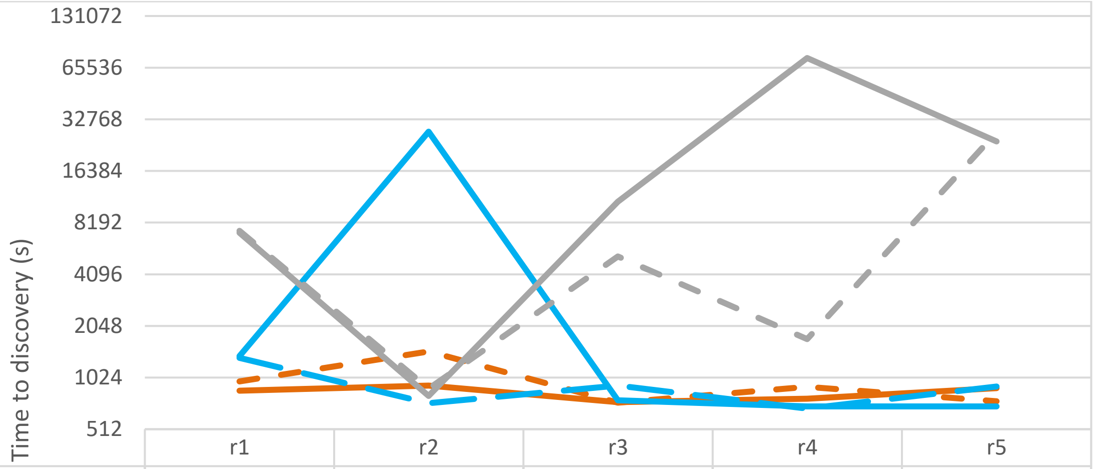


	r1	r2	r3	r4	r5
N1m1s	895	825	1692	2654	1199
V1m1s	658	792	638	727	998
N2m	4534	1223	4376	39397	19837
V2m	5068	1354	2767	11186	4310

Figure 16 Case 2 read_utmp discovery

Figure 16 showcases the random nature of fuzzing quite well. From this figure we can see that in most cases (60%) virtual execution has found read_utmp faster than in native execution. Orange lines represent one-master configuration and shows that read_utmp was found faster with a slave instance present in all runs compared to two-master instance.

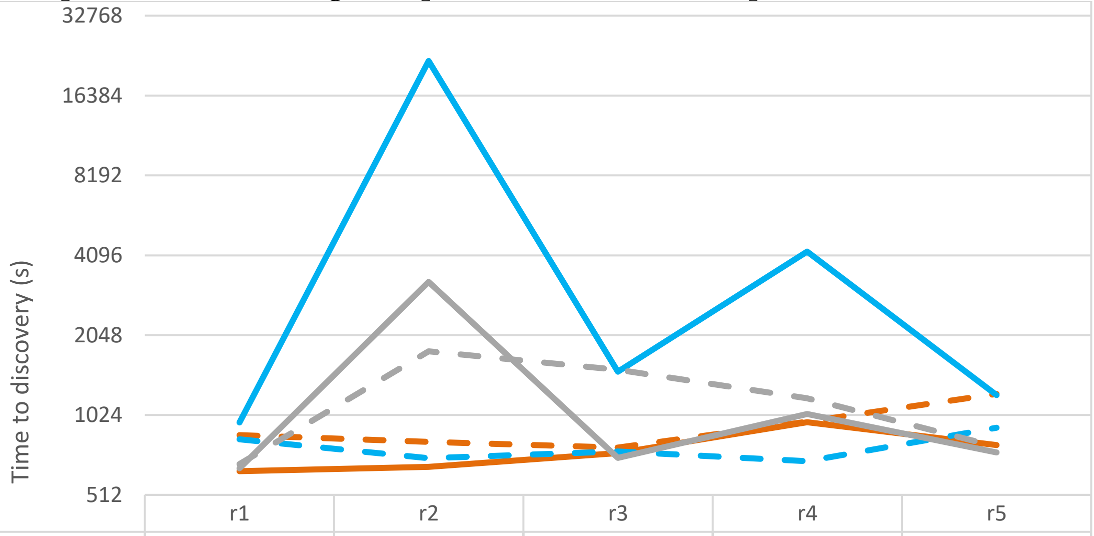
Below on Figure 17 the situation changes as first three-master instance is introduced. Both virtual and native execution three-master instances discover the read_utmp bug in all but one run (r2) in magnitudes more time than other instances. One-master instance virtual and native context discover the bug in the first 25 minutes (1500 seconds), while two-master instance is just as effective barring r2 in which N2m1s instance took almost 7 hours 42 minutes to find read_utmp.



	r1	r2	r3	r4	r5
N1m2s	858	919	736	771	889
V1m2s	971	1453	731	902	746
N2m1s	1363	27751	753	696	695
V2m1s	1333	725	918	675	909
N3m	7147	800	10847	74577	24340
V3m	7343	882	5211	1712	27509

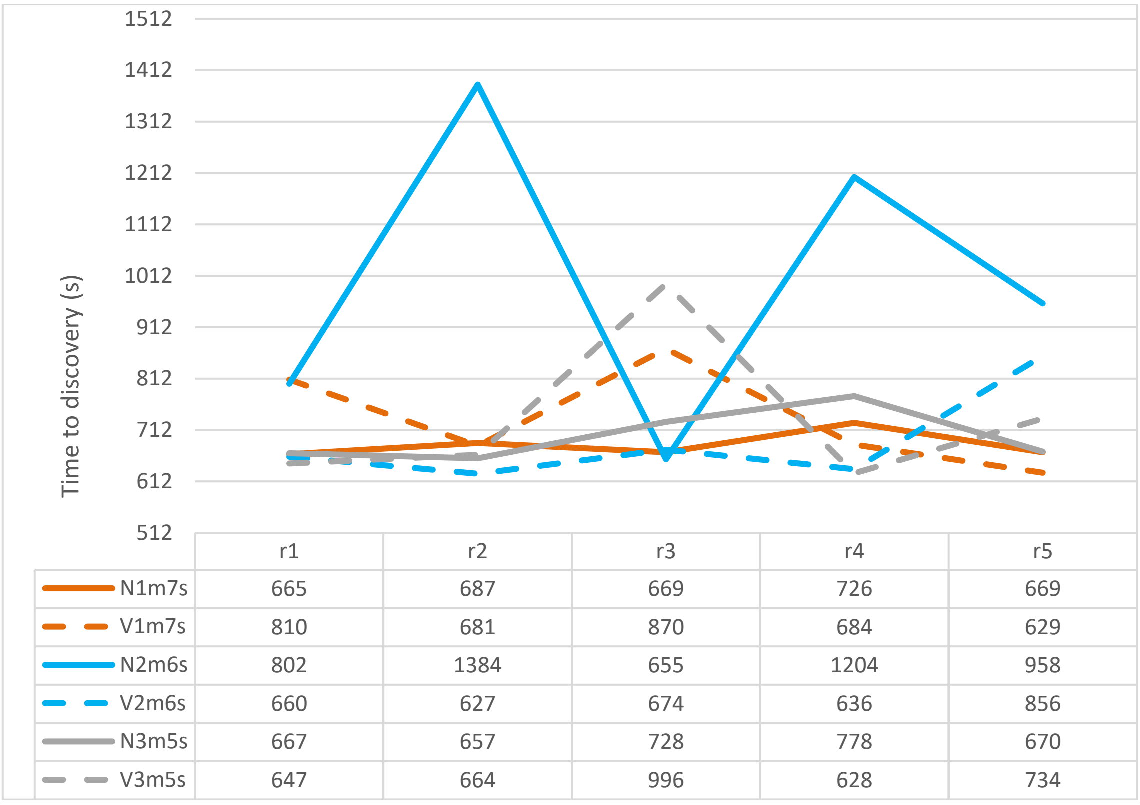
Figure 17 Case 3 read_utmp discovery

Below in Figure 18 one-master instance shows quite even discovery speeds in both native and virtual context. Judging from this figure the two- and three-master native execution instances have a large deviation compared to their virtual counterpart. Further testing is required and done in Chapter 5



	r1	r2	r3	r4	r5
N1m3s	629	654	736	962	789
V1m3s	859	811	772	967	1230
N2m2s	960	22118	1493	4230	1219
V2m2s	829	704	747	686	917
N3m1s	645	3251	706	1032	741
V3m1s	669	1780	1515	1186	778

Figure 18 Case 4 read_utmp discovery



	r1	r2	r3	r4	r5
N1m7s	665	687	669	726	669
V1m7s	810	681	870	684	629
N2m6s	802	1384	655	1204	958
V2m6s	660	627	674	636	856
N3m5s	667	657	728	778	670
V3m5s	647	664	996	628	734

Figure 19 Case 5 read_utmp discovery

Above in Figure 19 two-master instance still shows large dispersion in discovery speed in native execution. One-master instance still seems to find the read_utmp bug faster on average on both contexts, but is heavily contested by other configurations, including two-master virtual instance.

Following figures of 20 and 21 show the discovery time as a box plot in their own contexts. One-master instances are colored in shades of purple, two-master in shades of teal and three-master in shades of red. In these plots' lower discovery (i.e. bar height) is better, just as in Figures 16-19.

Below on Figure 20 the trend of discovery of read_utmp bug in native context is to be faster as cores are added, although it cannot be said that all runs, and instances follow this trend. However, in all cases the eight-core instance has found the read_utmp bug earlier than two-core instance, which could mean that speedup occurs. Between three and four core instances the results from this figure are inconclusive and require further analysis.

Same trend of inconclusiveness continues on Figure 21 far below. In general, the two-core instance still is beaten by eight-core instances. Furthermore, virtual instances perform better as 22 of 28 instances in native execution take 1000 or more seconds to find read_utmp, where in virtual execution only same number is 17 out of 27 (31r4 still missing).

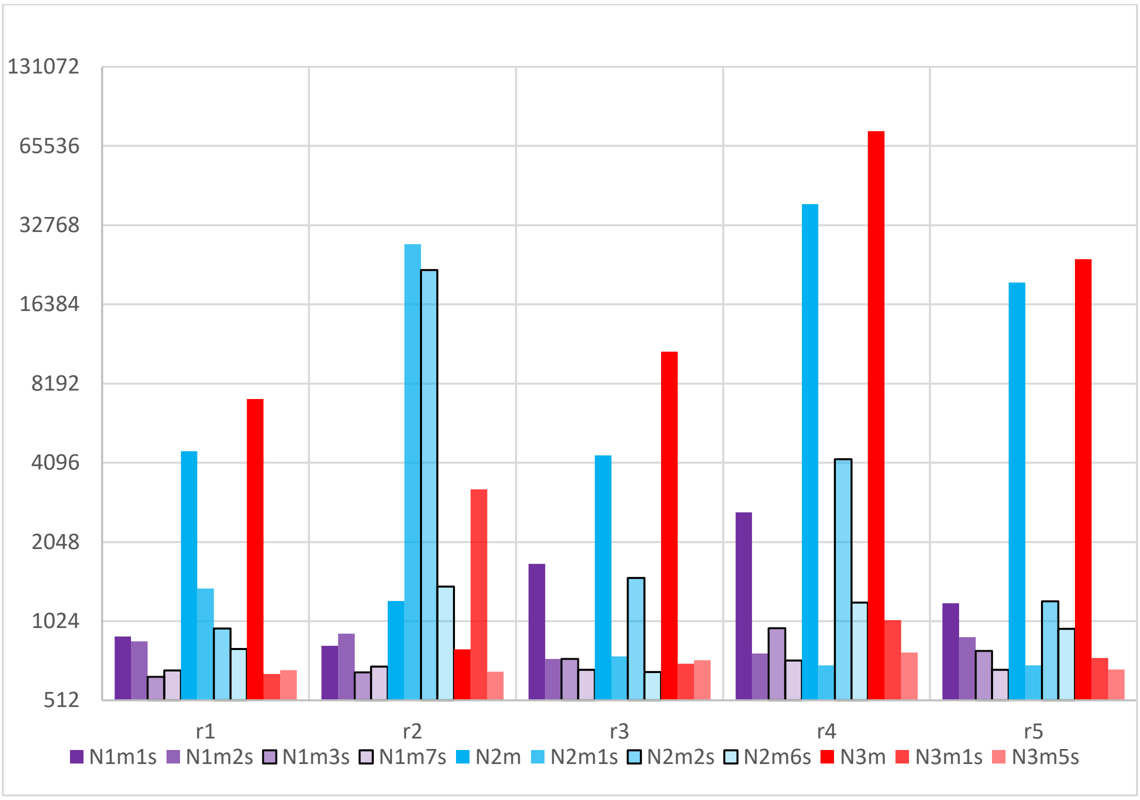


Figure 20 Native context read_utmp discovery

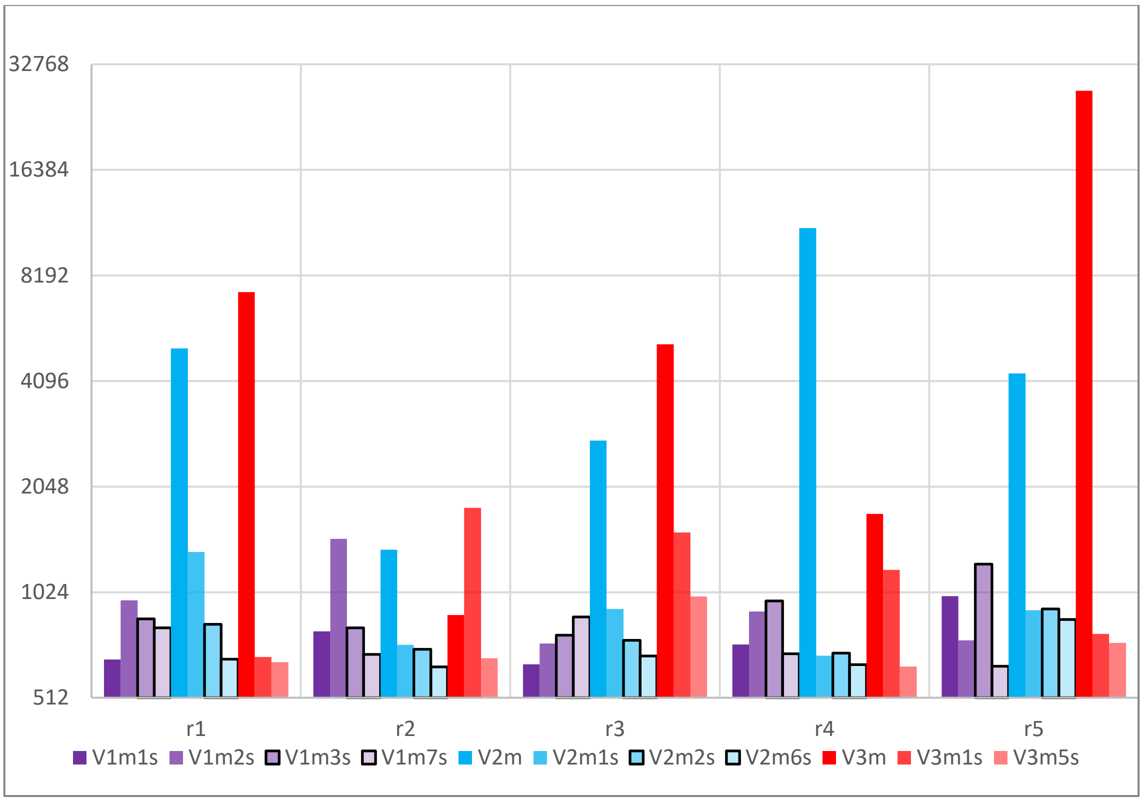


Figure 21 Virtual context read_utmp discovery

5 Discussion

Data from chapter 4 does not adequately answer the research questions laid out in Chapter 1. Therefore, further analysis is done in order to approve of disprove hypotheses from Chapter 3.3.3 and answer the research questions chapter by chapter, metric by metric. As a final chapter this we must discuss how this data should be interpreted and were we successful on answering research questions. Furthermore, there are few additional questions that this study does not answer and these information needs are addressed in the final chapter of this study.

5.1 Determining best configuration within cases

As per chapter 3.3.3 we can determine the best configuration within a case. This is done through comparing population value ranking with U-test. U-test is done in order to confirm or disapprove a hypothesis though examining the p-value (below 0,05) of test but in case study multiple sources of data must be investigated before passing judgement (Yin, 2014, p. 61). Arcuri and Briand suggest using Vargha Delaneys $\hat{A}_{12}$ standard effect size test to measure the difference between said populations. The hypotheses for this research question were:

h30: There is no difference in performance
h31: One-master configuration does perform better than other two configurations
h32: Two-master configuration does perform better than other two configurations
h33: Three-master configuration does perform better than other two configurations

To answer this question Cases 2-5 are used. Case 1 is non configurable and therefore is not used in this comparison. It would be possible to compare Greybox and Blackbox fuzzing efficiencies, but it is out of scope for this study.

5.1.1 Cumulative execution speed

In the following tables 25-28 we have used U-test to count the p-value between two populations: Population 1 is the results of one instance, while Population 2 are the combined results of two other instances. In this way we can test if a single configuration is statistically superior to rest of the case population. In addition, we have reported the $\hat{A}_{12}$ in percentage. $\hat{A}_{12}$ is always counted from the Population one (U1 in U-test). If no statistical difference is apparent, we can proceed to test *h31-h33* between two best instances in said case, which is done in the last column of table reporting the populations used, p values and Â12. Tables include both contexts and usable metric (CAES and read_utmp).

Table 25 Case 2 Cumulative Execution U-test and Â12.

	Native	Virtual
Population 1	1m1s	1m1s
Population 2	2m	2m
p	0,0182	0,0182
Â12	4 %	4 %

From Table 25 above the p-value of 0,04 is below the threshold of 0,05 which means that there is statistical difference between 1m1s and 2m in both contexts (contradicting the *h30* hypothesis). Furthermore, the effect size (Â12) in both cases indicates that 2m instance dominates over 1m1s 96% of the ranks. In short when cumulative execution speed is considered as a metric in case 2 the 2m instance is faster.

Table 26 Case 3 Cumulative Execution U-test and Â12.

	Native execution			
Population 1	1m2s	2m1s	3m	2m1s
Population 2	2m1s, 3m	1m2s, 3m	1m2s,2m1s	3m
p	0,1333	0,1500	0,0750	0,0909
Â12	32 %	36 %	82 %	20 %
	Virtual execution			
Population 1	1m2s	2m1s	3m	1m2s
Population 2	2m1s, 3m	1m2s, 3m	1m2s,2m1s	3m
p	0,1750	0,1250	0,1583	0,2182
Â12	58 %	30 %	62 %	48 %

From Table 26 above we it is apparent that there is no statistical difference between different configurations (confirming hypothesis *h30*), but their effect sizes varies from which we can furthermore test 2m1s and 3m in native context and 1m2s and 3m in virtual cases. There is no statistical difference in this case either, but the effect size measurement in native execution favors 3m and in virtual context only slightly favors 3m. Therefore, in Case 3 the most efficient configuration can be extrapolated to be 3m in native context. Virtual context the best configuration would also be 3m, as it wins the comparison between its rival by 2% and holds most effect size over other configurations (62%).

Table 27 Case 4 Cumulative Execution U-test and $\hat{A}_{12}$.

Native execution

Population 1	1m3s	2m2s	3m1s	2m2s
Population 2	2m2s, 3m1s	1m3s, 3m1s	1m2s,2m2s	3m1s
p	0,1583	0,1917	0,1417	0,1455
$\hat{A}_{12}$	38 %	46 %	66 %	32 %

Virtual execution

Population 1	1m3s	2m2s	3m1s	2m2s
Population 2	2m2s, 3m1s	1m3s, 3m1s	1m2s,2m2s	3m1s
p	0,1238	0,1905	0,1238	0,2000
$\hat{A}_{12}$	29 %	56 %	68 %	45 %

The trend of statistical indetermination continues in Case 4 as *h30* is confirmed to be true as no instance breaches the threshold of 0,05. However, the effect size measurement concludes that in both contexts 2m2s and 3m1s instances merit further investigation. There is not statistical difference between said instances, and 3m1s is marginally better in native execution while in virtual execution they are practically just as efficient. It can be concluded that in Case 4 there is no clear winner, but 2m and 3m instances fare better than their 1m counterpart favoring the 3m instance because of its larger effect size.

Table 28 Case 5 Cumulative Execution U-test and $\hat{A}_{12}$.

Native execution

Population 1	1m7s	2m6s	3m5s	2m6s
Population 2	2m6s, 3m5s	1m7s, 3m5s	1m7s, 2m6s	3m5s
p	0,1583	0,1750	0,1250	0,1636
$\hat{A}_{12}$	38 %	42 %	70 %	36 %

Virtual execution

Population 1	1m7s	2m6s	3m5s	1m7s
Population 2	2m6s, 3m5s	1m7s, 3m5s	1m7s, 2m6s	3m5s
p	0,1583	0,1000	0,1500	0,1091
$\hat{A}_{12}$	62 %	24 %	64 %	24 %

In Case 5 hypothesis *h30* is confirmed. Effect size measurement reveals that there might be statistical differences between 2m6s and 3m5s instances in native context, and between 1m7s and 3m5s in virtual context. Further testing for these cases reveals no statistical difference, but 3m5s instance has more effect size than rival instance. In conclusion for Case 5: there is no statistical difference, but 3m instance seems to fare better when compared to other instances.

In this metric the only statistical difference was found in Case 2 where in both contexts 2m instance was profoundly faster than its 1m counterpart. In further cases the results for statistical difference were inconclusive, but effect size measurements favored the 3m instance.

5.1.2 Shared bugs (read_utmp)

As per previous chapter, the following tables 29 - 32 provide information about the results of U-test and $\hat{A}_{12}$. Tables follow same formula laid down in previous chapter.

Table 29 Case 2 read_utmp U-test and $\hat{A}_{12}$.

	Native	**Virtual**
Population 1	1m1s	1m1s
Population 2	2m	2m
p	0,036	0,0000
$\hat{A}_{12}$	96 %	96 %

Results from U- and effect-size test are completely opposite when comparing Tables 29 and 25 (in previous chapter). There is statistical difference, so *h30* can be discarded. Furthermore, it seems that h31 holds true, as the statistical difference is apparent in 1m1s versus 2m (sub 0,05) and its effect size is 96%. Therefore, it can be concluded that when bug discovery is concerned the 1m instance fares better in Case 2 on both contexts.

Table 30 Case 3 read_utmp U-test and $\hat{A}_{12}$.

		Native		
Population 1	1m2s	2m1s	3m	1m2s
Population 2	2m1s, 3m	1m2s, 3m	1m2s,2m1s	2m1s
p	0,1417	0,1333	0,0667	0,1636
$\hat{A}_{12}$	66 %	68 %	16 %	64 %
		Virtual		
Population 1	1m2s	2m1s	3m	1m2s
Population 2	2m1s, 3m	1m2s, 3m	1m2s,2m1s	2m1s
p	0,1500	0,1083	0,0500	0,1818
$\hat{A}_{12}$	64 %	74 %	12 %	40 %

From table 30 the U-test is inconclusive on native context but disproves *h30* in virtual context (3m p-value < 0,05). In native context the effect size measurement favors 1m2s and 2m1s, which are not statistically clearly different (p-value of 0,16) but their effect size indicates that 1m2s would be faster most of the time in native context to find read_utmp. However, in virtual context it is apparent that 3m does not have large effect size, which concludes that it is the worst performing instance with statistical difference. Therefore, in virtual context the contest is between 1m2s and 2m1s where is no statistical difference but effect size slightly favors 2m1s. From Case 3 read_utmp results the best instance (though not clearly) for native execution is 1m2s and 2m1s for virtual execution.

Table 31 Case 4 read_utmp U-test and $\hat{A}_{12}$.

Native

Population 1	1m3s	2m2s	3m1s	1m3s
Population 2	2m2s, 3m1s	1m3s, 3m1s	1m2s,2m2s	3m1s
p	0,0833	0,0417	0,1667	0,1636
$\hat{A}_{12}$	80 %	10 %	60 %	64 %

Virtual

Population 1	1m3s	2m2s	3m1s	2m2s
Population 2	2m2s, 3m1s	1m3s, 3m1s	1m2s,2m2s	3m1s
p	0,1524	0,1143	0,1524	0,1556
$\hat{A}_{12}$	36 %	73 %	40 %	65 %

As can be seen on table 31 above Case 4 exhibits statistical difference in native execution (2m2s p-value of 0,0417) disapproving *h30*, but not in virtual context. In native context the 2m2s is statistically different, but its effect size is low meaning that it is the worst performing instance. Therefore, on native execution 1m3s and 3m1s are investigated and while not having clear difference (statistically) the effect size favors 1m3s. From assessing the effect sizes of virtual instances, it is apparent that 1m3s has the least impact to population, so 2m2s and 3m1s are tested. From U-test the result is inconclusive, but effect size measurement favors 2m2s. Therefore, in Case 4 native execution the best instance considering discovery of read_utmp is 1m3s while in virtual context it is 2m2s.

Table 32 Case 5 read_utmp U-test and $\hat{A}_{12}$.

Native

Population 1	1m7s	2m6s	3m5s	1m7s
Population 2	2m6s, 3m5s	1m7s, 3m5s	1m7s, 2m6s	3m5s
p	0,1333	0,0833	0,1583	0,2000
$\hat{A}_{12}$	68 %	20 %	62 %	56 %

Virtual

Population 1	1m7s	2m6s	3m5s	2m6s
Population 2	2m6s, 3m5s	1m7s, 3m5s	1m7s, 2m6s	3m5s
p	0,1417	0,1417	0,2083	0,1818
$\hat{A}_{12}$	34 %	66 %	50 %	60 %

Table 32 above shows that there is no statistical difference in bug discovery in either contexts, confirming *h30*. The effect size measurements in native context favor 1m7s and 3m5s while in virtual context 2m6s and 3m5s. In native context this further analysis is quite inconclusive as the 1m7s is no clearly better (56 effect size). Virtual context shows similar results for 2m6s. As all the tests do not show a clear best instance, it can be concluded that there is major difference between performances, but native execution slightly favors 1m7s and virtual 2m6s instances.

5.1.3 Conclusion

The results from Tables 25-32 and their supporting decision making has been compressed to following table 33 below. In this table it is shown if there was statistical difference between metrics and what was the conclusion regarding assessing the best instance for said case.

Best configuration for maximizing execution speed seems to be configuration that has the most master instances. Only clear difference (statistically) is from Case 2, but all other Cases do not proclaim a clear winner and their results are determined from effect size comparison. Furthermore, it seems that the execution speed trend in effect size starts to homogenize, which is most probably due to fact that the slave instances have less execution speed than master instances in average (for further information check Appendix 2).

Table 33 Compressed information for assessing best configuration

	CAES	*p*	**$\hat{A}_{12}$**	**Read_utmp**	*p*	**$\hat{A}_{12}$**
Case 2N	2m	0,0182	96 %	1m1s	0,0364	96 %
Case 2V	2m	0,0182	96 %	1m1s	0,0000	96 %
Case 3N	3m	0,0750	82 %	1m2s	0,1417	66 %
Case 3V	3m	0,1417	66 %	2m1s	0,1083	74 %
Case 4N	3m1s	0,1417	66 %	1m3s	0,0833	80 %
Case 4V	3m1s	0,1238	68 %	2m2s	0,1143	73 %
Case 5N	3m5s	0,1250	70 %	1m7s	0,1333	68 %
Case 5V	3m5s	0,1500	64 %	2m6s	0,1417	66 %

From the bug discovery speed perspective it seems that in native context the less master instances is present, the faster bug is discovered (remembering that read_utmp was the first bug to be discovered by all instances) and that in virtual context the two-master approach works better. As in previous metric the only statistically clear case was Case 2, while all other cases use effect size measurement to decipher the best configuration.

In conclusion there is no clear best combination for different cases. For execution speed it might be best to use as many master instances as you can while for bug discovery use one- or two-masters depending on the context the fuzzers are running.

Execution speed might be a good metric when trying to make an fuzzer faster (as done by (Xu, Kashyap, Min, & Kim, 2017)), but optimizing your configuration for execution speed seemingly hampers the bug discovery potential of AFL when run for 24 hours. For these short runs it seems to be better to concentrate to one- or two-masters (depending on context) to find crashes faster. After all, fuzzers job is to find crashes which can be filtered for bugs and vulnerabilities, not just to run at peak execution speed.

5.2 Performance difference between native and virtual execution

Same tests are done to determine the performance gap between virtual and native context: U-test is done to determine if there is clear statistical difference and $\hat{A}_{12}$ is used to determine the extent of this difference. The tests are done pairwise for each case and usable metric. Each table 34 – 37 in the following chapter 5.2.1 is constructed the same way as in chapter 5.1 tables but the population sizes are smaller as this is a pairwise testing and not testing against the rest of case population and the populations are marked "N" for native- and "V" for virtual case population. Hypotheses for these tests are the same as deducted in chapter 3.3.3 for Research question 2:

h20: There is no difference in performance considering N and V executions
h21: There is no difference in performance in contexts in case one
h22: There is no difference in performance in contexts in case two
h23: There is no difference in performance in contexts in case three
h24: There is no difference in performance in contexts in case four
h25: There is no difference in performance in contexts in case five

5.2.1 U-test and Effect size

Table 34 Case 1: one core used

	Execution speed	**read_utmp**
Population 1	1 AFL N (18)	1 AFL N (7)
Population 2	1 AFL V (18)	1 AFL V (18)
p	0,058558559	0,061538462
$\hat{A}_{12}$	12,04 %	15,87 %

One core instance does not show statistical difference in execution speed nor in read_utmp. It does however heavily favor the virtual execution as the effect size for native instance is below 20%. There is difference in performance so *h20* is disproven and so is *h21* as effect size measurement favors virtual execution.

Table 35 Case 2: two-cores used

	CAES		**read_utmp**	
Population 1	1m1s N	2m N	1m1s N	2m N
Population 2	1m1s V	2m V	1m1s V	2m V
p	0,01818182	0,036363636	0,036364	0,163636
$\hat{A}_{12}$	4,00 %	8,00 %	8,00 %	36,00 %

Table 35 above shows previously mentioned test results from Case 2 from both virtual and native context. U-test results in all comparisons indicate that there is performance difference which disproves *h20* and *h22*. From the effect sizes we can deduce that in general native execution is slower and has used more time to find read_utmp than the virtual context. Only test that does not show statistical difference is read_utmp 2m N vs 2m V where the effect size still favors the virtual context.

Table 36 Case 3: three-cores used

	CAES			**Read_utmp**		
Population 1	1m2s N	2m1s N	3m N	1m2s N	2m1s N	3m N
Population 2	1m2s V	2m1s V	3m V	1m2s V	2m1s V	3m V
p	0,07272727	0,109090909	0,181818	0,181818	0,2	0,163636
$\hat{A}_{12}$	16,00 %	24,00 %	40,00 %	60,00 %	44,00 %	36,00 %

From Table 36 only statistical difference is found in 1m2s Native and Virtual context in Cumulative Average Execution Speed. As only minority of this case show statistical difference the *h23* cannot outright be disproven. As per Case 2 (Table 35 above) the effect sizes still favor the virtual context but only in CAES metric as the read_utmp favors native execution on one master instance and somewhat favors virtual context when masters are added. *h23* is disproven as there is performance difference between the instances, but the performance difference scale tips different way with different configurations.

Table 37 Case 4: four-cores used

	CAES			**Read_utmp**		
Population 1	1m3s N	2m2s N	3m1s N	1m3s N	2m2s N	3m1s N
Population 2	1m3s V	2m2s V	3m1s V	1m3s V	2m2s V	3m1s V
p	0,09090909	0,072727273	0,022222	0,072727	0	0,177778
$\hat{A}_{12}$	20,00 %	16,00 %	5,00 %	84,00 %	0 %	60,00 %

For Case 4 (Table 37 above) the three-master instance disproves the *h24* in CAES and two-master instance fortifies this resolution for read_utmp. Effect size comparison still favors virtual execution in CAES but for read_utmp favors native execution except in two-master instance. *h24* Is disproven as both metrics show at least one statistical difference and their effect sizes are not inconclusive (50%). Same kind of trend of configuration differences emerge as for Case 3 (Table 35 above) as CAES favors virtual execution but for bug discovery the difference varies between configurations.

Table 38 Case 5: eight-cores used

	CAES			Read_utmp		
Population 1	1m7s N	2m6s N	3m5s N	1m7s N	2m6s N	3m5s N
Population 2	1m7s V	2m6s V	3m5s V	1m7s V	2m6s V	3m5s V
p	0,12727273	0,163636364	0,2	0,163636	0,072727	0,181818
$\hat{A}_{12}$	28,00 %	64,00 %	56,00 %	64,00 %	16,00 %	40,00 %

Finally, for Case 5 (Table 38) no statistical difference is found through U-testing. Effect size comparison reveals that for CAES native execution is only slightly favored for two- and three-master instances while having almost an decisive difference towards virtual execution for one-master instance. For bug discovery the roles are reversed as two- and three-master instances favor virtual execution and one-master instance favors native execution, but the differences are small. As there is not statistical difference in any of the tests and the differences in effect sizes are small the *h25* is not disproven and stands firm proclaiming that there is no difference in performance between virtual and native context in Case 5, which uses eight cores for computing.

5.2.2 Conclusions

The cases examined disprove all but one hypothesis. Therefore, it can be said that there is performance difference between native and virtual context up until four usage of four cores. The difference is not clear cut as used configuration has impact on how the scale tips and how much. However, in general it can be deduced that virtual execution favors execution speed while native context favors bug discovery. It would be interesting to examine the point where the execution starts to "level-out", but it is out of scope for this study.

5.3 Determining speedup and scalability between instances

From chapter 2.2.2 the scalability was function of speedup. Speedup is counted from latency of one worker divided by latency of many workers. If this speedup is below one it is sublinear, and above one superlinear while one being linear speedup. Efficiency is counted by dividing speedup by the number of used workers and denotes if performance was lost (below one) or gained (above one) by adding more workers.

In this chapter we have used the data from Case 1 from both contexts stored in Appendix 3. This data is used to count the latency for speedup. Metric CAES yields information in "executions per/sec" which can be converted to latency by raising it to the power of -1. Shared bug metric yields the latency outright and can be used as is for counting speedup. For easy reference we have included this tabulated data in appendix four in order not to convolute chapters.

Following sub-chapters (5.3.2 – 5.3.5) tables and figures all follow the same form for easy comparisons. In Tables speedup and efficiency are calculated per configuration (up on top of table). On Figures the dotted line denotes derived values of efficiency while bard graphs denote speedup values and efficiency (dotted line) axis is on the right of picture while speedup (bar graph) is read from the left. Different colors are used on figures to differentiate the min and max values.

5.3.1 Testing hypotheses

In order to start analyzing the results of we test the hypothesizes from chapter 3.3.3. Data from Case 1 (Appendix 1) is and the data CAES and read_utmp from Chapters 4.4.2 and 4.5.1 respectfully. $\hat{A}_{12}$ Effect size test measures the effect size as done in previous Chapters 5.1 and 5.2 while U-test are done in order to test for hypotheses:

h10: There is no statistical difference in instance performance
h11: Two-core instance is better than one core instance
h12: Three-core instance is better than one core instance
h13: Four-core instance is better than one core instance
h14: Eight-core instance is better than one core instance

Following two tables below (Tables 39 and 40) show the test results. Tables hold both CAES and Read_utmp metrics in their respective areas. Table 39 shows the results form native context and Table 40 shows virtual context. Population 1 for both tests is the Case 1 and population 2 is marked in both tables.

Table 39 Native context Cases 2-5 compared to Case 1

	CAES										
	1m 1s	**2m**	**1m 2s**	**2m 1s**	**3m**	**1m 3s**	**2m 2s**	**3m 1s**	**1m 7s**	**2m 6s**	**3m 5s**
Rsum	276	276	276	276	276	276	276	276	276	276	276
U1	156	156	156	156	156	156	156	156	156	156	156
U2	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	0	0	0	0	0	0	0	0
$\hat{A}_{12}$	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %
	read_utmp										
	1m 1s	**2m**	**1m 2s**	**2m 1s**	**3m**	**1m 3s**	**2m 2s**	**3m 1s**	**1m 7s**	**2m 6s**	**3m 5s**
Rsum	78	78	78	78	78	78	78	78	78	78	78
U1	13	14	13	14	20	13	13	13	13	13	13
U2	0	1	0	1	7	0	0	0	0	0	0
p	0	0,0128	0	0,0128	0,0897	0	0	0	0	0	0
$\hat{A}_{12}$	0,00 %	2,86 %	0,00 %	2,86 %	20,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %

From Native context results (Table 39 above) it is apparent that population one (Case 1, one core instance) is statistically inferior to all cases in both metrics. Only case where U-test does not show clear-cut results is read_utmp 3m instance and in this rivalry Case 1 effect size is only 20%. Considering these results *h10* is revoked and *h11-h14* are proven to be true. This means that it is possible to use said metrics to measure scalability between instances as both metrics show improvement to Case 1 in native context.

Table 40 Virtual context Cases 2-5 compared to Case 1

	CAES										
	1m 1s	**2m**	**1m 2s**	**2m 1s**	**3m**	**1m 3s**	**2m 2s**	**3m 1s**	**1m 7s**	**2m 6s**	**3m 5s**
Rsum	276	276	276	276	276	276	276	253	276	276	276
U1	156	156	156	156	156	156	156	161	156	156	156
U2	0	0	0	0	0	0	0	0	0	0	0
p	0	0	0	0	0	0	0	0	0	0	0
$\hat{A}_{12}$	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %
	read_utmp										
	1m 1s	**2m**	**1m 2s**	**2m 1s**	**3m**	**1m 3s**	**2m 2s**	**3m 1s**	**1m 7s**	**2m 6s**	**3m 5s**
Rsum	276	276	276	276	276	276	276	253	276	276	276
U1	156	161	156	156	167	156	156	161	156	156	156
U2	0	5	0	0	11	0	0	0	0	0	0
p	0	0,0181	0	0	0,0399	0	0	0	0	0	0
$\hat{A}_{12}$	0,00 %	5,56 %	0,00 %	0,00 %	12,22 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %	0,00 %

Virtual context (Table 40 above) shows the same trend on results as Table 39 on native context. The results for virtual context revoke the *h10* and confirm *h11-14*. Therefore, we can measure the speedup in virtual context the same way as in native context.

5.3.2 Native execution metric: CAES

Table 41 Tabulated speedups from CAES in native execution

		1m 1s	**2m**	**1m 2s**	**2m 1s**	**3m**	**1m 3s**	**2m 2s**	**3m 1s**	**1m 7s**	**2m 6s**	**3m 5s**
speedup	**min**	2,12	2,35	3,13	3,13	3,33	3,51	3,83	3,95	7,19	7,40	7,40
	max	1,72	1,91	2,58	2,65	2,77	3,25	3,28	3,25	6,11	5,72	6,49
efficiency	**min**	1,06	1,18	1,04	1,04	1,11	0,88	0,96	0,99	0,90	0,93	0,92
	max	0,86	0,95	0,86	0,88	0,92	0,81	0,82	0,81	0,76	0,72	0,81

Table 41 (above) shows the speedup and efficiency values derived from CAES of all native cases. Speedup values are all above one showing superlinear speedup across the configurations. Efficiency however does not follow this trend as efficiencies around one are reported with minor increase in 2m and 3m instances. From this can be concluded that while speedup is gained it is less and less efficient to use more cores while measuring execution speed. This deduction only solidifies the other conclusions regarding execution speed (Chapter 4.4, 5.1 and 5.2) that the masters have better execution speed but adding slaves to configuration skews the execution speed downward.

Figure 22 (below) visualizes values from Table 41 and it is apparent that speedup jumps from four cores to eight cores which is not surprising as the metric sums up the execution speed of configuration. Efficiency is shown to mainly be below 1 which has been discussed on previous paragraph.

From Table 41 and Figure 22 it is possible to deduce that execution speed does rise (as expected) when more workers are used. This is concurrent with work done by Xu et al. (Xu, Kashyap, Min, & Kim, 2017) and would be interesting to see if the results follow similar pattern of efficiency drop around 15 workers.

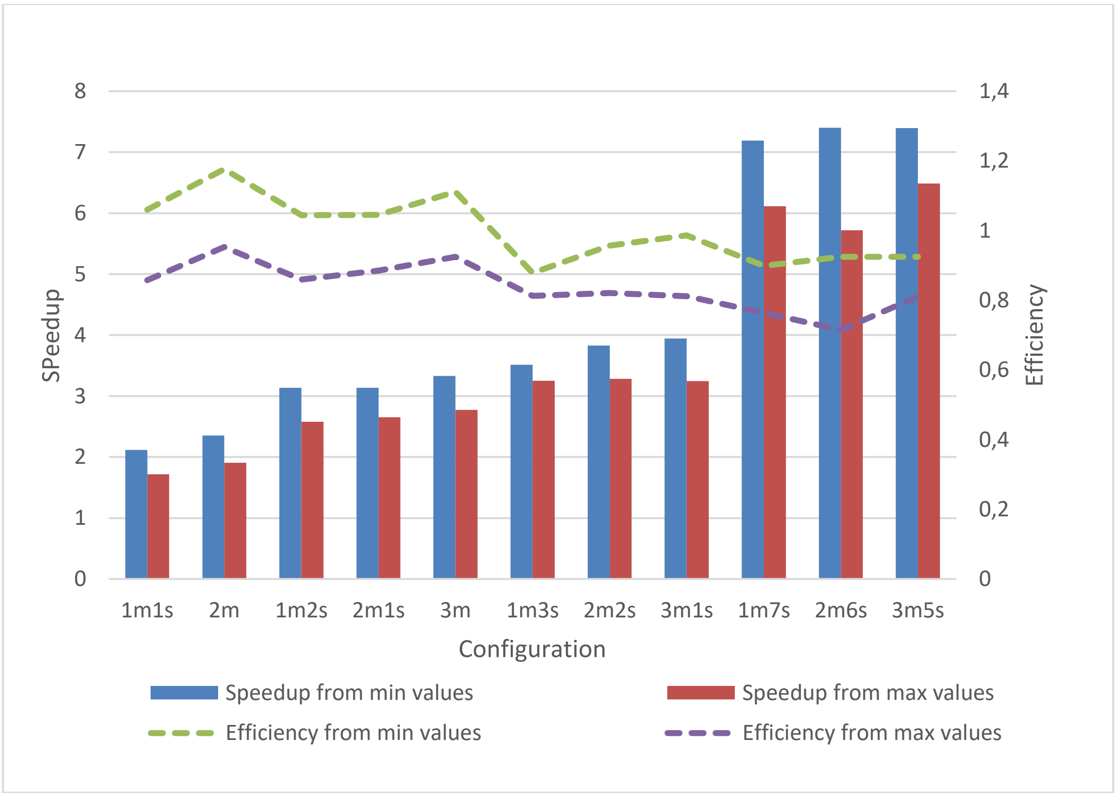


Figure 22 Bar- and line-graph of Table 40 values (CAES)

It is highly possible that execution speed-based metrics do not measure the whole instance scalability as well as shared bug metric. Execution speed is the speed of loop execution (*Throughput*) between TCG and Monitor fuzzer components

(Chapter 2.1.2 Figure 3) while the fuzzers job is to find crashes and not just execute as fast as it can. Execution is also fuzzing target based and in this case the target is small yielding large execution speed. Measuring the scalability of execution speed can be done, but the results might not represent the scalability of results rather the scalability of different components of fuzzer instance.

5.3.3 Native execution metric: shared bug (read_utmp)

Table 42 Tabulated speedups from read_utmp in native execution

		1m 1s	**2m**	**1m 2s**	**2m 1s**	**3m**	**1m 3s**	**2m 2s**	**3m 1s**	**1m 7s**	**2m 6s**	**3m 5s**
speedup	**min**	22,18	22,18	36,86	39,03	33,91	43,13	28,26	42,06	40,79	41,42	41,29
	max	16,21	1,87	79,96	2,65	0,99	76,39	3,32	22,60	101,2	53,09	94,45
efficiency	**min**	11,09	11,09	12,29	13,01	11,30	10,78	7,06	10,51	5,10	5,18	5,16
	max	8,10	0,93	26,65	0,88	0,33	19,10	0,83	5,65	12,65	6,64	11,81

Comparing Tables 41 (previous chapter) and 42 above it is apparent that shared bug metric shows larger speedup and efficiency values. As in other cases where these two metrics are compared side-by side they differentiate largely on cases where configuration only consists of masters. In this case the all master instances show superlinear scaling, but their efficiency is below zero. Any instance that has an slave-instance shows superlinear scaling and good efficiency numbers.

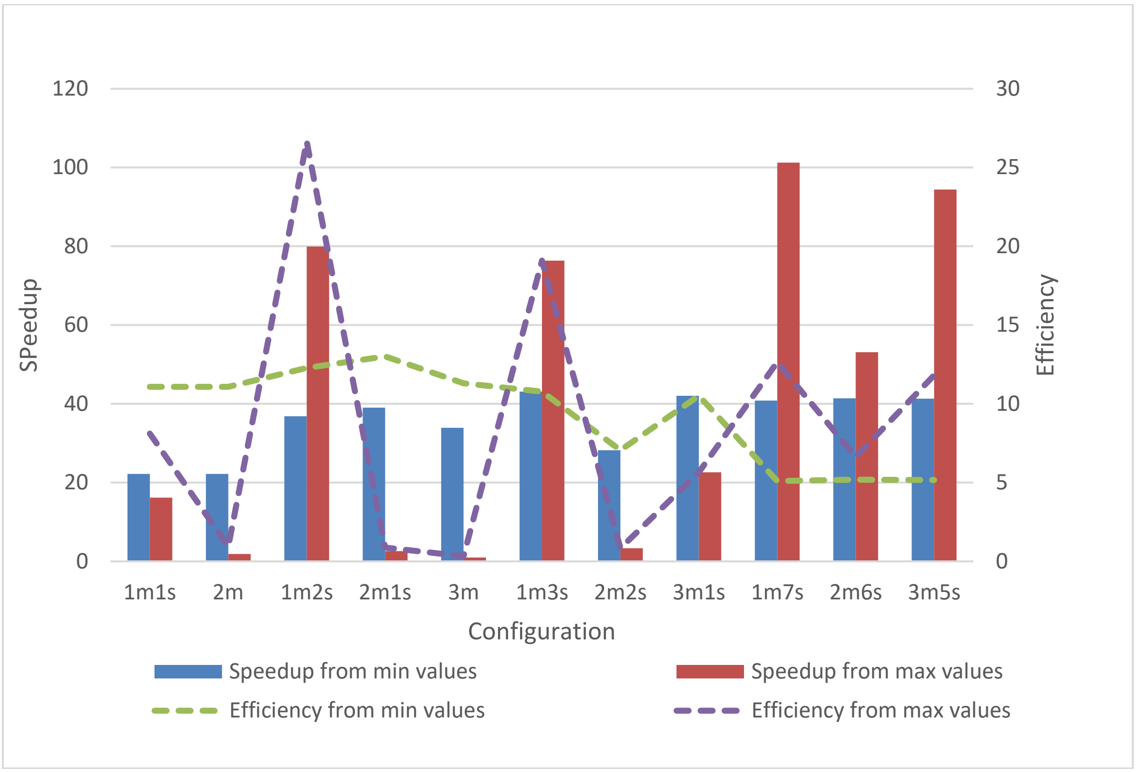


Figure 23 Bar- and line-graph of Table 41 values (read_utmp)

Figure 23 shows that the minimum values of speedup are quite consistent varying between 20-40 times speedup. Large speedup max values are directly related to the max values of one core instance (Appendix 3 table 1: 73483 seconds vs. Appendix 2 table 919-74577 s). Efficiency drops of max values (below 1) are shown for 2m, 2m1s, 3m and 2m2s which show that these instances do not gain efficiency as more slaves are added but the deviation of minimum values is not as great as max values. This would mean that fore mentioned instances have great variance in time to find the first crash in this case.

5.3.4 Virtual execution metric: CAES

Table 43 Tabulated speedups from CAES in virtual execution

		1m 1s	2m	1m 2s	2m 1s	3m	1m 3s	2m 2s	3m 1s	1m 7s	2m 6s	3m 5s
speedup	**min**	1,83	1,99	2,52	2,65	2,65	2,98	3,32	3,53	6,01	5,71	5,72
	max	1,77	1,95	2,56	2,51	2,55	3,18	3,32	3,21	6,25	5,95	6,27
efficiency	**min**	0,92	1,00	0,84	0,88	0,88	0,75	0,83	0,88	0,75	0,71	0,72
	max	0,88	0,97	0,85	0,84	0,85	0,79	0,83	0,80	0,78	0,74	0,78

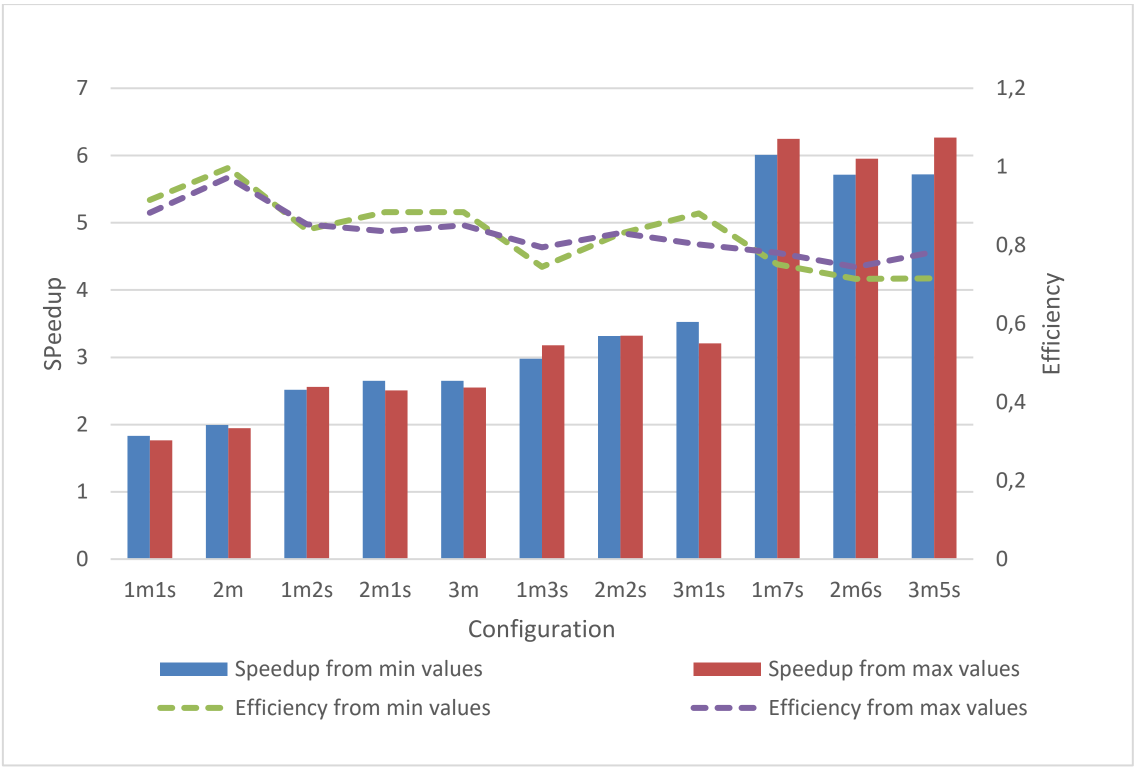


Figure 24 Bar- and line-graph of Table 40 values (CAES)

Table 43 and its tabulated speedup shows that virtual execution has less dispersion between efficiency values (the dotted lines are closer to each other on all cases), but the speedup is not quite as when comparing eight core instances. As the effectiveness is in all cases below zero, it can be said that in virtual context adding more workers yield speedup in general, but the effectiveness (i.e. relative speedup) decreases as more workers are included. These results follow the same trend as native execution results in Chapter 5.3.2

5.3.5 Virtual execution metric: shared bug (read_utmp)

Table 44 Tabulated speedups from read_utmp in virtual execution

		1m 1s	**2m**	**1m 2s**	**2m 1s**	**3m**	**1m 3s**	**2m 2s**	**3m 1s**	**1m 7s**	**2m 6s**	**3m 5s**
speedup	**min**	3,36	3,36	6,23	6,75	5,16	5,90	6,64	6,81	7,24	7,26	7,25
	max	12,86	5,83	44,87	48,91	2,37	53,00	71,09	36,63	74,94	76,16	65,46
efficiency	**min**	1,68	1,68	2,08	2,25	1,72	1,48	1,66	1,70	0,91	0,91	0,91
	max	0,84	0,84	0,69	0,75	0,57	0,37	0,41	0,43	0,11	0,11	0,11

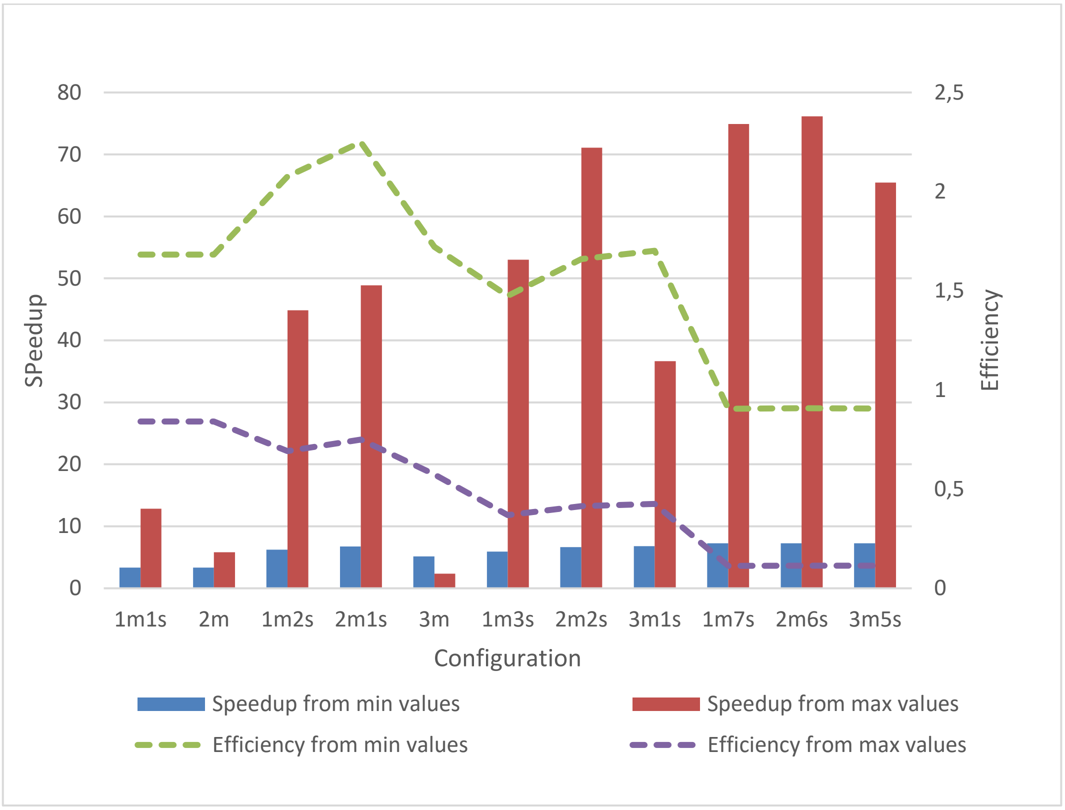


Figure 25 Bar- and line-graph of Table 41 values (read_utmp)

The speedup and efficiency values of virtual context shared bug (read_utmp) are shown on Table 44 and plotted to Figure 25. From the figure 25 efficiency has a downward trend and speedup is increased as workers are added. Speedup is not linear per configuration as the two- and three-master instances together with 3m1s show lower speedup than other same worker count instances. Most efficient instance in virtual context is 2m1s with efficiency ranging from 0,75 to 2,25.

5.4 Discussion

In this chapter we consolidate the information from previous chapters. Information gained from testing AFL was plentiful and provided opportunities for analyzation. From dataset two rivals were formed: cumulative average of execution speed (CAES) and the time to find a bug that was shared between all cases in this multiple case study.

Two of the metrics discussed in Chapter 4 are not used, as they did not provide adequate dispersion within the metric. The metrics in question are code coverage and unique bug count. Coverage was measured with AFL and did not provide enough fine-grained information to be usable as a metric. If Code coverage was to be used as a metric for performance testing, measuring it with percentage loses precision. Instead of Code Coverage the number of paths taken could be used, but then the paths would need to be identified somehow (so not to overcount paths) and would require additional framework which could induce overhead to the fuzzer.

Unique bug count was deemed unfit as most of the instances only found a single bug and therefore the cases do not have dispersion that would answer research questions. Unique bug count is well used metric in fuzzing (Klees, Ruef, Cooper, Wei, & Hicks, 2018), but in this multiple-case study the randomness of fuzzers did not yield enough information. In future research if this metric is used as performance measure, the fuzzer should make longer- and more runs for this metric to be usable. From literature Arcuri and Briand´s 24h and 5 runs minimum (Arcuri & Briand, 2011) are not enough for this metric to work.

5.4.1 Best configuration

Testing in chapter 5.1 was done case-by case determining if there is statistical difference in performance when measured metrics are CAES and read_utmp-bug. For CAES the only statistical difference was found from Case2 where 1m1s instance was superior in both contexts. Read_utmp had more dispersion on its results and statistical differences were found in Cases 2 -4. Further Effect size analysis revealed that for CAES master instances were favored and for Read_utmp slaves respectfully.
A key observation for this question is, that CAES suffers from datapoint disparity between master and slave instances, which starts to skew metric towards homogenous results as more slave workers are added. This skewing is not apparent in read_utmp, the shared bug metric.

In conclusion, the fuzzers job is to find crashes that can be filtered to bugs and further evaluated for vulnerabilities. As CAES is throughput of instance, it does not measure the results of fuzzer. As read_utmp effectively is a race condition for finding a shared bug (shallow one in this case) it measures the whole fuzzers performance. As for the question of best configuration the answer is that for each instance configuration start filling slave instances before masters. As the first slave is added master the difference between instances becomes apparent. After adding the first slave the results more or less start intertwining and therefore after the first slave the best configuration cannot be explicitly confirmed and should be decided on case by case basis: if more deterministic checking is needed a master should be used and if seed files or specifications span a large area of the input space a slave could be better to unload the input queue.

Switching the configuration “in flight” would be an interesting research subject from performance point of view but would need additional metrics that measure these “in flight” datapoints as read_utmp only measures the results of a single run and CAES could lead to wrong conclusions on whole fuzzer performance.

5.4.2 Performance difference

Performance difference was measured in Chapter 5.2 on pairwise comparison between virtual and native context cases. Statistical difference was found on both Cases 1 and 2 while effect size measurement favored virtual execution, exception being Case 2 2m configuration that favored virtual execution only slightly after effect size measurement (36 % on native, 64 %on virtual). As for further cases the answers is more complicated.

When using three or four cores the difference in virtual and native execution is more configuration and metric based. CAES favors virtual execution but is not statistically better than native execution. For read_utmp one-master instance favors native execution while-three master instance favors virtualization in all master configuration (3m) and native execution with one slave added (3m1s). For two-master configuration the results are more polarized as on Case 3 the execution environments are even while for case 4 native execution environment trumps virtual completely statistically and with effect size.

The same trend of configuration and metric disparity continue to Case 5. For eight core instances no statistical differences are found, but effect size varies between metric and configuration. For one-master instances CAES favors virtual execution while read_utmp favors native execution slightly. Two-master instances favor native in CAES and virtual in read_utmp, the exact opposite of one master instance. Three-master instances are more in between, not clearly favoring any environment as no statistical difference is found and their effect sizes are 56% favoring native in CAES and 40% favoring virtual for read_utmp.

As said in chapter 5.2.2 it could be generalized that CAES favors virtual execution and read_utmp favors native execution. In this multiple-case study we did not examine a target that could destabilize the system, which would be the case when fuzzing system and kernel modules. Therefore, as the performance is not hugely impacted while using multiple cores it would be prudent to use virtual execution instead of native as it proves isolation. Furthermore, is small core count of one to three cores are used virtualization should be used as it seems to fare better than it native counterpart armed with the information laid out in previous chapter.

5.4.3 Scalability of AFL

As stated in Chapter 2.2 *speedup* and *efficiency* are measured in order to measure if program is scalable. *Efficiency* is calculated by dividing *speedup* by the number of workers used for said task, which measures how much more efficient the instance is compared to one worker instance.

Scalability can be assessed from both metrics of speedup and efficiency we have plotted the efficiency of one-, two- and three-master instances in following Figures 26-28 for both metrics of CAES and read_utmp. In these figures we have calculated Amdahl´s and Gustafsson-Barsis´s law equivalent of serialized portion of work and their efficiencies. These helper lines are added in order to visualize the serialized portion outlined in both laws (McCool, Robinson, & Reindeer, 2012, pp. 59-62) and decide if configuration adheres to either of said theories. As a generalization of both laws, the less serial work is done the better scalability the program has.

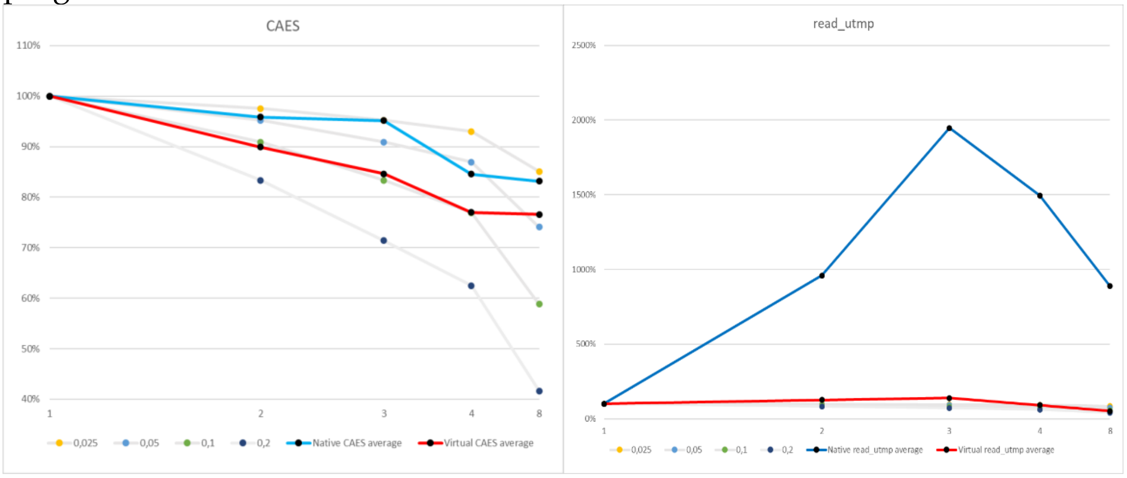


Figure 26 one-master instance efficiency

Figure 26 shows native execution metrics as blue while red shows virtual execution metrics. On the left metric used for efficiency is CAES while on the right it is latency to read_utmp. CAES is quite linear on its efficiency and drops downward as workers are added which is concurrent with previously mentioned theories. On the right read_utmp native execution does not follow the presumption of losing effectiveness as it peaks at three cores used with almost 2000% increase in effectiveness when compared to a single core instance, but this can be explained through the native instance having massive difficulties finding read_utmp (see appendix 3) and therefore has multitudes more latency than the virtual execution. Virtual effectiveness for read_utmp stays above 100% for two and three core instances, but dips below 100% for rest of the cases. Still it can be said, that both environments scale exceedingly well on two and three core instances and lose only small amount of effectiveness as workers are added, or even doubled when comparing four and eight core instances. For precise comparison and data revisit chapter 5.3.3 and 5.3.5.

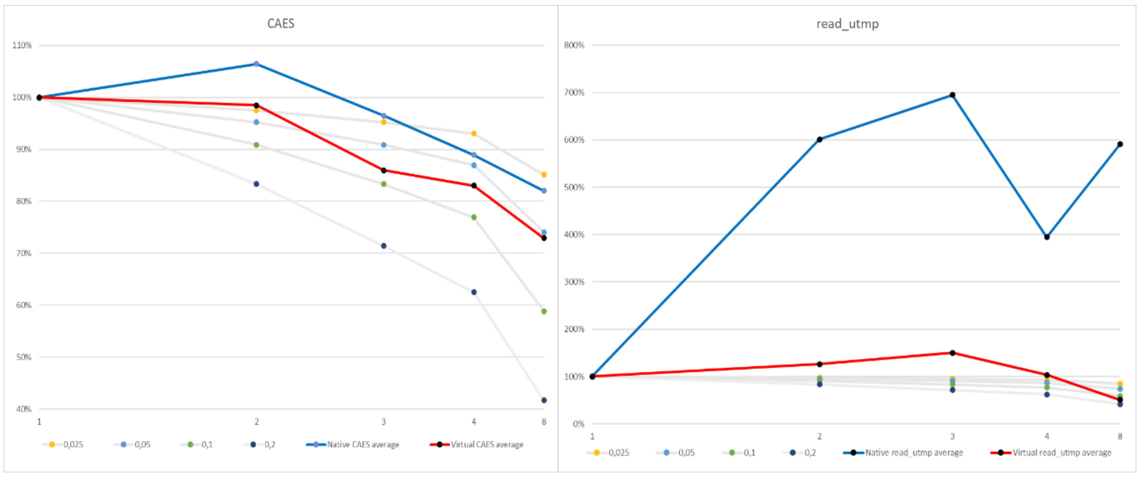


Figure 27 two-master instance efficiency

Figure 27 follows the same schema than Figure 26 far above. It would seems that CAES works at peak efficiency compared to one core instance when two-masters are used, but starts losing it after an slave is added. Read_utmp native execution continues to report exceedingly good values, while virtual execution is more modest values but still showing 100% exceeding values on cases 2,3 and 4 (See chapter 5.3.4 for precise values). For read_utmp adding a master seems to increase the effectiveness in virtual configuration, while making the native execution less effective. Again, fuzzer shows remarkable efficiency when adding more workers and could be said that scales well when four or less cores are used with two-master configuration.

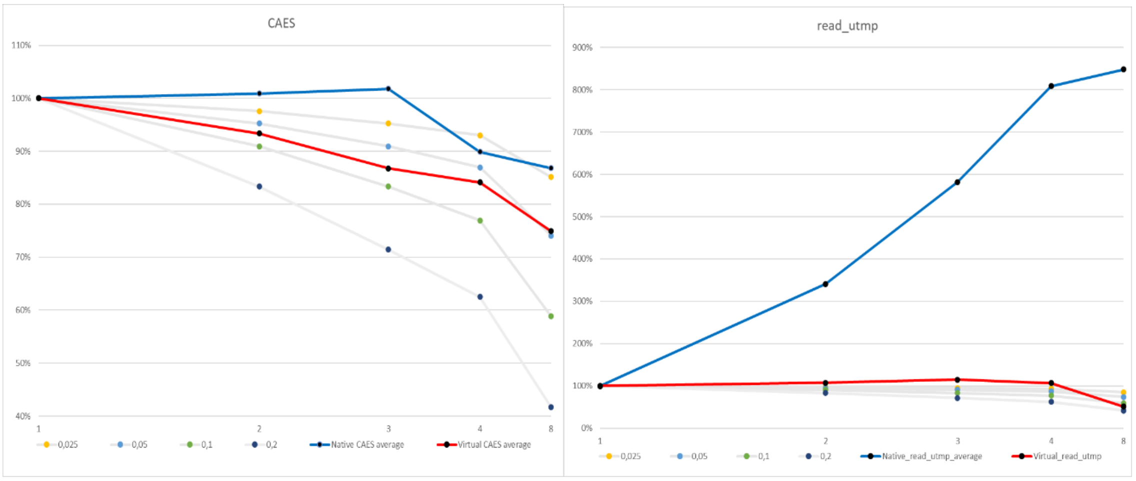


Figure 28 three-master instance efficiency

As per previous observations in Figures 26 and 27 (far above) the CAES follows same trend of losing performance when more slaves are added in native execution while virtual execution loses efficiency more linearly. For read_utmp native execution the efficiency rises as slaves are added (three masters is the intended maximum), but virtual efficiency follows performance theory quite well.

Data in this case is not homogenous enough to say that native execution would scale better than virtual execution. As only 7 / 18 instances of native execution found read_utmp while on virtual environment 18/18 instances were successful. The figures however do show correlation between used cores, for example in Figures 26 and 27 efficiency rises all the way to three cores used and the plummets on both environments. Again, this is a question of configuration rather than efficiency.

In conclusion AFL scales well when one or more slaves are used when the results are considered as a metric for performance. Scaling is strong as adding workers reduces the time to find a bug and therefore follows Amdahl´s law. From this above 100% efficiency we can also deduce, that with low core counts there is little-to-none serialized work being done and the performance overhead starts incurring after three cores are being used impacting the performance and lowering efficiency below 100%.

5.4.4 Further research

In a sense this research has failed to use all the metrics available to asses performance differences both in configuration and execution environment. Therefore, it would be prudent to repeat this study with more execution runs and time to see if metrics laid out on this study can be used in performance review.

Performance theory itself remains a good subject when combined with randomized algorithms. Amdahl´s- and Gustafsson-Barsis´s law are crude instruments when compared to work-span model (McCool, Robinson, & Reindeer, 2012, pp. 62-65). This would require defining the critical path to a crash and how it was achieved alongside with the information on how many the input was mutated. A task that possibly requires modifying fuzzer data-gathering either by modifying the fuzzer itself to provide this information or using a taint-tracing framework in conjunction with fuzzer outputted information.

Finally, as the last proposition: this should all be done at run-time. In a sense it would be effective to be able to configure and run instances based on demand. This would help the fuzzer for example modify a slave instance to master instance when coverage stalls. Modern fuzzers do use different types of problem-solving techniques like concolic execution and taint tracing to help this checking, but are these effective approaches? Do they scale well? In order to find optimal approaches to solutions the performance must still be tested. No one likes doing work that has no meaning, not even fuzzers.

6 References

Arcuri, A., & Briand, L. (2011). ICSE ´11 . *A Practical Guide for Using Statistical Tests to Asses Randomized Algorithms in Software Engineering.* Waikiki, Honolulu, USA.

Ball, T. (2003). *Abstraction-guided Test Generation: A Case study.* Redmond: Microsoft Research.

Bell, G. C. (1967). *Time Shared Computers.* Pittsburg: Carnegie Mellon University.

Bellard, F. (2005). USENIX Annual Technical Conference. *QEMU, a Fast and Portable Dynamic Translator* (pp. 41-46). USENIX Association.

Bradbury, D. (2018, 12 3). *Faster fuzzing ferrets out 42 fresh zero-day flaws.* Retrieved 12 12, 2018, from Naked security: https://nakedsecurity.sophos.com/2018/12/03/faster-fuzzing-ferrets-out-42-fresh-zero-day-flaws/

Böhme, M., Pham, V.-T., & Roychoudhury, A. (2017). Transactions on software engineering . *Coverage-Based Greybox Fuzzing as Markov Chain.*

Carlton, A. (2017, 11 19). *AFL-Docker.* Retrieved 1 14, 2020, from https://github.com/AlexandreCarlton/afl-docker

Cha, S. K., Averinos, T., Rebert, A., & Brumley, D. (2012). IEEE Symposium on Security and Privacy. *Unleashing MAYHEM on Binary Code* (pp. 380-394). San Francisco: IEEE.

Chipunov, V., Kuznetsov, V., & Candeae, G. (2011). ASPLOS ´11. *S2E: A Platform for In-Vivo Multi-Path Analysis for Software Systems* (pp. 1-14). Newport Beach: ACM.

Coldewey, D. (2016, 8 6). *Carnegie Mellon's Mayhem AI takes home $2 million from DARPA's Cyber Grand Challenge.* Retrieved 1 20, 2020, from Techcrunch : https://techcrunch.com/2016/08/05/carnegie-mellons-mayhem-ai-takes-home-2-million-from-darpas-cyber-grand-challenge/

Deja Vu Security. (2014, 2 23). *Peach Community Edition, Peach-pit.* Retrieved 11 14, 2019, from http://community.peachfuzzer.com/v3/PeachPit.html

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., . . . Whelan, R. (2016). 2016 IEEE Symposium on Security and Privacy. *LAVA: Large-scale Automated Vulnerability Addition.*

Domas, C. (2018). *Project Rosenbridge: Hardware Backdoors in x86 CPUs.* Las Vegas: DEFCON 26.

Dong , W. H., & Hsien-Hsin , L. S. (2008). Extending Amdahl's law for Energy-Efficient computing in the Many-Core Era. *IEEE Computer Society*, 24-31.

Drewry, W., & Ormandy, T. (2007). *Flayer: Exposing Application Internals.* Retrieved 3 1, 2019, from Usenix web site: https://www.usenix.org/legacy/event/woot07/tech/full_papers/drewry/drewry_html/index.html

Dunbar, D.;Engler, D.;& Cadar, C. (2008). 8th USENIX Symposium on Operating Systems Design and Implementation. *KLEE: Unassisted and Automatic*

Generation of High-Coverage Tests for Complex Systems Programs (ss. 209-224). Stanford University.

Finnish Ministry of Defence. (2015, 6 10). *KATAKRI 2015 - Information security audit tool for authorities.* Retrieved 2 12, 2019, from https://www.defmin.fi/en/administrative_branch/defence_security/katakri_2015_-_information_security_audit_tool_for_authorities

Fratric, I. (2016, 7 7). *Github Project Zero WinAFL*. Retrieved 2 11, 2019, from A fork of AFL for fuzzing Windows binaries : https://github.com/googleprojectzero/winafl

Fraze, Dustin. (2020, 1 20). *Cyber Grand Challenge*. Retrieved 1 20, 2020, from DARPA Archive: https://www.darpa.mil/program/cyber-grand-challenge

Gamozo Labs. (2018, 9 16). Scaling AFL to a 256 thread machine. Retrieved from https://gamozolabs.github.io/fuzzing/2018/09/16/scaling_afl.html

Ganesh, V., Leek, T., & Rinard, M. (2009). *Taint-based Directed Whitebox Fuzzing.* Massachusets: MIT Computer science and Artificial Intelligence lab.

Github. (2018, 6 6). *Cyber Grand Challenge*. Retrieved 1 20, 2020, from https://github.com/CyberGrandChallenge/

Godefroid, P., Klarlund, N., & Sen, K. (2005). PLDI´05. *DART: Directed Automated Random Testing* (p. 213223). Chicago: ACM.

Google. (2019, 2 1). *Google Open Source Project OSS-Fuzz*. Retrieved 2 11, 2019, from https://opensource.google.com/projects/oss-fuzz

Google. (2019, 2 15). *Google Scholar: Case Study Research*. Retrieved 2 15, 2019, from https://scholar.google.fi/scholar?hl=fi&as_sdt=0%2C5&q=Case+Study+research+Yin&btnG=

Google. (2019, 2 15). *Google Scholar: Research Methods for Business Students*. Retrieved 2 15, 2019, from https://scholar.google.fi/scholar?hl=fi&as_sdt=0%2C5&q=Research+methods+for+business+students&btnG=

Google LLC. (2017, 8 7). *Github: AFL parallel fuzzing.* Retrieved December 19, 2019, from https://github.com/mirrorer/afl/blob/master/docs/parallel_fuzzing.txt

Google LLC. (2019, 11 10). *Github: Google AFL.* Retrieved 11 15, 2019, from Afl-fuzz.c: https://github.com/google/AFL/blob/master/afl-fuzz.c

Gustafson, J., & Barsis, E. (1988). Communications of the ACM. *Reevaluating Amdahl's Law.* ACM.

Haller, I., Slowinska, A., Neugschwandtner, M., & Bos, H. (2013). Proceedings of the 22nd USENIX Security Symposium. *Dowsing for overflows: A guided fuzzer to find buffer boundary violations* (pp. 49-63). Washington D.C: USENIX Association.

Hill, M. D., & Marty, M. R. (2008). IEEE Computer Society. *Amdahl's Law in the Multicore Era*, 33-38.

Hongliang, L., Xiaoxiao, P., Xiaodong, J., Wuweu, S., & Jian, Z. (2018). IEEE Transactions on Reliability, VOL. 67, NO. 3, September 2018. *Fuzzing: State of the Art*, (pp. 1199 - 1218).

ISO/IEC/IEEE. (2013). 29119 part 1: Concepts and Definitions. Switzerland.

ISO/IEC/IEEE. (2015, 12 1). ISO/IEC/IEEE 29119 part 4: test techniques. Switzerland.

Kaksonen, R., Laakso, M., & Takanen, A. (2000, 1 1). Vulnerability Analysis of Software through Syntax Testing. Oulu, Finland.

Klees, G., Ruef, A., Cooper, B., Wei, S., & Hicks, M. (2018). CCS '18. *Evaluating Fuzz Testing.* Toronto, Ontario, Kanada.

Kovari, A., & Dukan , P. (2012). 2012 IEEE 10th Jubilee International Symposium on Intelligent Systems and Informatics. *KVM & OpenVZ virtualization based IaaS Open Source Cloud Virtualization Platforms: OpenNode, Proxmox Ve* (pp. 335 - 339). Subotica: IEEE.

Krishnamoorthy, S., Hsiao, M. S., & Lingappan, L. (2010). 2010 19th IEEE Asian Test Symposium. *Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs.* Shanghai: IEEE.

Larson, E., & Austin , T. (2003). Proceedings of the 12th USENIX Security Symposium. *High Coverage Detection of Input-Related Security Faults* (pp. 121-136). Michigan: Advanced Computer Architecture Laboratory.

Lattner, C., & Adve, V. (2004). *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* Illinois: University of Illinois at UrbanaChampaign.

Laumann, O., Davison, W., Weigert, J., & Schroeder, M. (2019, 9 18). *Man Screen.* Retrieved 2 25, 2020, from Linux Man pages: https://linux.die.net/man/1/screen

Liang, J., Wang, M., Chen, Y., Jiang, Y., & Zhang, R. (2018). SANER 2018. *Fuzz Testing in Practice: Obstacles and Solutions.* Campobasso, Italia.

Linn, A. (2017, 7 21). *AI for security: Microsoft Security Risk Detection makes debut.* Retrieved 2 11, 2019, from Microsoft blogs: https://blogs.microsoft.com/ai/ai-for-security-microsoft-security-risk-detection-makes-debut/

Linux Foundation. (2019, 8 2). *Linux Man pages: Exec.* Retrieved 11 15, 2019, from http://man7.org/linux/man-pages/man3/exec.3.html

McClure, S., Scambray, J., & Kurtz, G. (2012). *Hacking exposed 7.* New York: McGraw-Hil cop.l.

McCool, M., Robinson, A. D., & Reindeer, J. (2012). *Structured parallel programming : patterns for efficient computation.* Amsterdam, Boston,Mass: Elsevier/Morgan Kaufmann 2012.

Microsoft. (2006). The Security Development Lifecycle . In M. Howard, & S. Lipner, *Enough is Enough: The Threats have Changed* (pp. 3-13). Microsoft Press.

Microsoft. (2019). *Microsoft SDL: Security engineering resource list.* Retrieved 2 11, 2019, from https://www.microsoft.com/en-us/securityengineering/sdl/resources

Miller, B. P., Fredriksen, L., & So, B. (1990). In Proceedings of the Workshop of Parallel and Distributed Debugging. *An Empirical Study of the Reliability of UNIX utilities.* Academic Medicine. Retrieved 3 1, 2019, from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.7648

Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A., & Steidl, J. (1995). *Fuzz Revisited: A Re-Examination of the Reliablity of Unix Utilities and Services.* Wisconsin: University Of Wisconsin.

Ministry of Finance Finland. (2013, 5 12). *Vahti-instructions.* Retrieved 2 12, 2019, from https://www.vahtiohje.fi/web/guest/5/2013-paatelaitteiden-tietoturvaohje

MITRE. (2020, 17 1). *MITRE CVE database*. Retrieved 1 17, 2020, from https://cve.mitre.org/index.html

Molnar, D., Li, X. C., & Wagner, D. A. (2009). SSYM'09 Proceedings of the 18th conference on USENIX security symposium. *Dynamic Test Generation To Find Integer Bugs in x86 Binary Linux Programs* (pp. 67 - 72). Montreal: USENIX Association Berkeley. Retrieved from Github: https://argp.github.io/public/50a11f65857c12c76995f843dbfe6dda.pdf

Nagy, B. (2018, 9 6). *Github: Crashwalk.* Retrieved 2 19, 2020, from https://github.com/bnagy/crashwalk

Nagy, S., & Hicks, M. (2019). 2019 IEEE Symposium on Security and Privacy. *Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing.* San Francisco: IEEE.

Nanda, S., Chiueh, T.-c., & Stony, B. (2005, 1). *A Survey on Virtualization Technologies.* Retrieved 1 8, 2020, from https://www.computing.dcu.ie/~ray/teaching/CA485/notes/survey_virtualization_technologies.pdf

National Institute of Standards and Technology. (2018, 1). *National Institute of Standards and Technology Computer Security Resource center - QEMU.* Retrieved 3 6, 2019, from https://csrc.nist.gov/glossary/term/QEMU

Nethercote, N., & Seward, J. (2003). Electronic Notes in Theoretical Computer Science 89 No. 2. *Valgrind: A Program Supervision Framework* (pp. 44-66). Cambridge: Elsevier Scienve B. V.

Nethercote, N., & Seward, J. (2007). ACM Sigplan notices vol 42. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation* (pp. 89-100). San Diego: ACM.

Paul, J. M., & Meyer, B. H. (2006). International Journal of Parallel Programming vol 35. *Amdahl's Law Revisited for Single Chip Systems* (pp. 101-123). Springer Science+Business Media LLC.

Pham, V.-T., Böhme, M., Andrew, S. E., Căciulescu, A. R., & Roychoudhury, A. (2019, 2 4). *Github AFLSMART.* Retrieved 2 11, 2019, from https://github.com/aflsmart/aflsmart

Pham, V.-T., Böhme, M., Santosa, A. E., Roychoudhury, A., & Câciulescu, A. R. (2018, 11 23). *Smart Greybox Fuzzing.* Retrieved 12 12, 2018, from Cornell University library: https://arxiv.org/abs/1811.09447

Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., & Bos, H. (2017). NDSS ´17. *VUzzer: Application-aware Evolutionary Fuzzing.* San Diego.

Rebert, A., Cha, S. K., Avgerinoss, T., Foote, J., Warren, D., Grievo, G., & Brumley, D. (2014). Proceedings of the 23rd USENIX Security Symposium. *Optimizing Seed Selection for Fuzzing* (pp. 861-875). San Diego: USENIX.

Saunders, M., Lewis, P., & Thornhill, A. (2012). *Research methods for business students.* Harlow, England: Pearson Education 2012.

Sen, K., Marinov, D., & Agha, G. (2005). ESEC-FSE´05. *CUTE: A Concolic Unit Testing Engine for C.* Lisbon: ACM.

Singh, A. (2004, 1). *An Introduction to Virtualization*. Retrieved 1 8, 2020, from http://www.kernelthread.com/publications/virtualization/

Srinagesh, K. (2006). *The Principles of Experimental Research.* Amsterdam ; Burlington: Elsevier/Butterworth-Heinemann.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., . . . Vigna, G. (2016). NDSS ´16. *Driller: Augmenting Fuzzing Through Selective Symbolic Execution.* San Diego.

Takanen, A., Demott, J. D., & Miller, C. (2008). *Fuzzing for Software Security Testing and Quality Assurance.* Artech House.

Ubuntu. (2020, 2 17). *Ubuntu installation and system requirements.* Retrieved 2 20, 2020, from https://help.ubuntu.com/community/Installation/SystemRequirement s

University of Oulu. (2017, 5 10). *Github / Cloudfuzzer*. Retrieved 2 11, 2019, from https://github.com/ouspg/cloudfuzzer

Van Tonder, R., Kotheimer, J., & Le Goues, C. (2018). Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18). *Semantic Crash Bucketing.* Montpellier: ACM.

Wang, J., Chen, B., Wei, L., & Liu, Y. (2017). 2017 IEEE Symposium on Security and Privacy. *Skyfire:Data-Driven Seed Generation for Fuzzing.*

Wikipedia Common. (2019, 9 7). *Wikipedia: Fork System Call* . Retrieved 11 15, 2019, from https://en.wikipedia.org/wiki/Fork_(system_call)

Wikipedia Commons. (2017, 10 5). *Wikipedia: Stack trace*. Retrieved 11 13, 2019, from https://en.wikipedia.org/wiki/Stack_trace

Woo, M., Cha, S. K., Goettlib, S., & Brumley, D. (2013). CCS'13. *Scheduling Black-box Mutational Fuzzing.* Berlin: ACM.

Xu, W., Kashyap, S., Min, C., & Kim, T. (2017). CCS '17. *Designing New Operating Primitives to Improver Fuzzing Performance.* Dallas, Texas: ACM.

Yin, R. K. (2014). *Case Study Research Design and methods.* SAGE Publications, Inc.

Zalewski, M. (2014, 24 12). *AFL plot data*. Retrieved 2 13, 2019, from http://lcamtuf.coredump.cx/afl/plot/

Zalewski, M. (2016). *American Fuzzy Lop (2.52b)*. Retrieved 2 11, 2019, from http://lcamtuf.coredump.cx/afl/

Zalewski, M. (2017, 11 5). *AFL QEMU Readme.* Retrieved 1 14, 2020, from Github: https://github.com/mirrorer/afl/tree/master/qemu_mode

Zalewski, M. (2017, 11 5). *AFL Readme.* Retrieved 2 12, 2019, from 7. Interpreting output: http://lcamtuf.coredump.cx/afl/README.txt

Zalewski, M. (2017, 11 5). *AFL technical details.* Retrieved 2 12, 2019, from http://lcamtuf.coredump.cx/afl/technical_details.txt

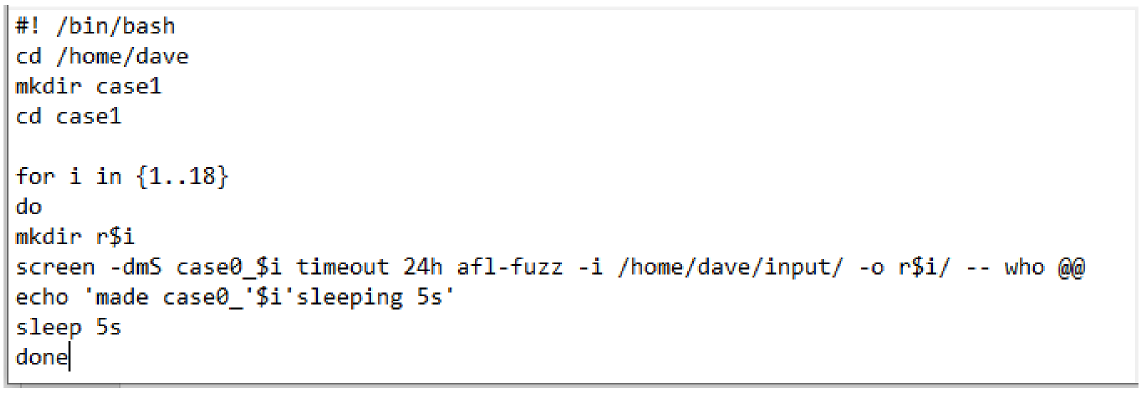
Zalewski, M. (2017, 11 4). *AFL-changelog*. Retrieved 11 7, 2018, from http://lcamtuf.coredump.cx/afl/ChangeLog.txt

APPENDIX 1: USED COMMAND LINE COMMANDS

Examples of scripts are shown in this appendix. The scripts itself are uploaded to github at:

https://github.com/Boring-Username/MSc-Thesis-AFL-Scalability-/

Case 1 script:

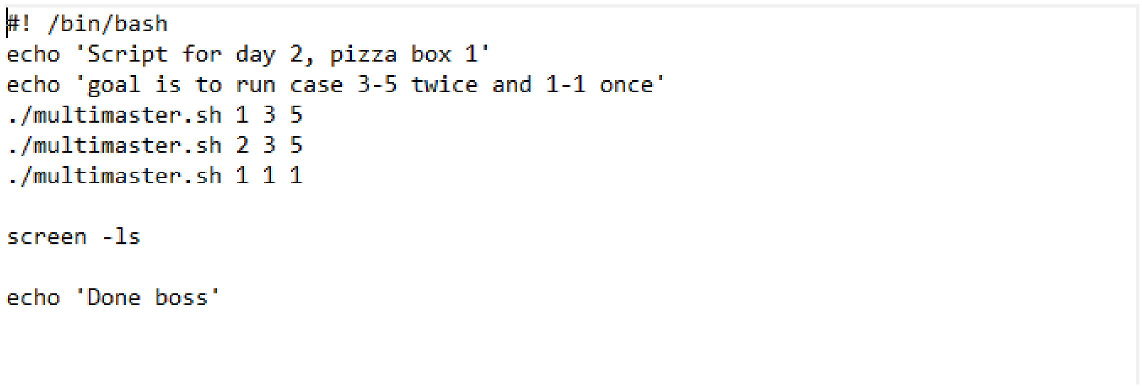


```
#! /bin/bash
cd /home/dave
mkdir case1
cd case1

for i in {1..18}
do
mkdir r$i
screen -dmS case0_$i timeout 24h afl-fuzz -i /home/dave/input/ -o r$i/ -- who @@
echo 'made case0_'$i'sleeping 5s'
sleep 5s
done
```

Script for launching configurations

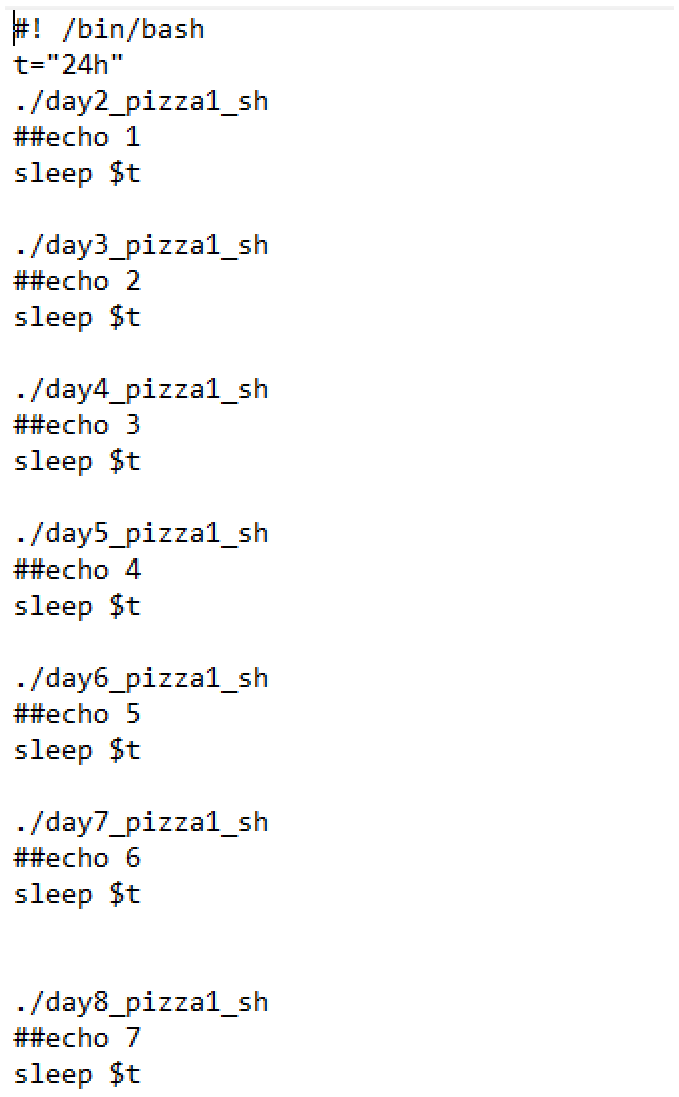
```
#! /bin/bash
echo usage 'script' 'run#' 'Masters#' 'Slaves#'
## setting some used variables, t for time and p for daves path
t='20s'
p='/home/dave'

##Check for all the arguments
## if to check for more than 4 (case is not fit to handle "gt" expression), case to check for correctness. Can be
expanded to include singular instances

if [ "$#" -gt "3" ]
then
    echo 'RTFM' ; exit 1
fi

case $# in
[1-2])
    echo 'RTFM' ; exit 1
;;
[3])
    echo 'Good BOY! Syntax is OK'
;;
esac

##Checking if dave is present
if [ -d $p ]
then
    echo "Dave exists, we can move forward"
else
    echo "Making dave"
    cd /home
    mkdir dave
fi

##traversing to dave
cd $p

## Checking if afl can be run (ie. no throttling and others). AFL returns 1 if something went wrong during startup
echo 'Checking AFL, Spooling up...'

timeout 2s afl-fuzz -i $p/input -o $p/output/ -- who @@ &> test.txt

if [ "$?" -eq "1" ]
then
    echo 'AFL is not feeling well, check test.txt'
    exit 1
else
    echo 'AFL seems fine'
    rm test.txt
fi

echo 'Case run number:'$1'. Master instances:'$2'. Slave instances:'$3'. Runtime is:'$t

##checking if case is present, if not, making one, traversing to it and preparing sync_dir for AFL
c="case_"$2$3r$1
if [ -d "/home/dave/"$c ]
then
    echo "Case stucture exists??? Exiting" ; exit 1
else
    mkdir $c
    cd $c
    mkdir sync_dir
fi

## make $2 amount of masters in a loop, 0 masters exits the script
if [ "$2" -eq "0" ]
then
    echo 'you need a master for instance, exiting' ; exit 1
fi

for ((m=1; m<="$2"; m++ ))
do
    screen -dmS $c"_Master"$m timeout $t afl-fuzz -i $p/input/ -o sync_dir -M "Master"$m":"$m"/"$2 -- who @@
    echo 'did '$m 'Master'
done
## check if slaves need to be made, and do the needfull
if [ "$3" -gt "0" ]
then
    for ((s=1; s<="$3"; s++ ))
    do
        echo 'did' $s 'Slave'
        screen -dmS $c"_Slave"$s timeout $t afl-fuzz -i $p/input/ -o sync_dir -S Slave$s -- who @@
    done
fi

 echo "Script done boss"
```

Example of launching multiple instances



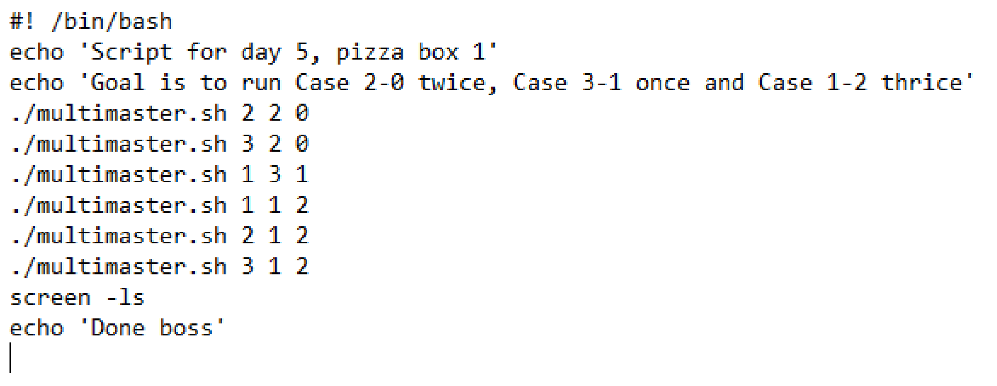
```
#! /bin/bash
echo 'Script for day 2, pizza box 1'
echo 'goal is to run case 3-5 twice and 1-1 once'
./multimaster.sh 1 3 5
./multimaster.sh 2 3 5
./multimaster.sh 1 1 1

screen -ls

echo 'Done boss'
```

Example of scripting multiple days:

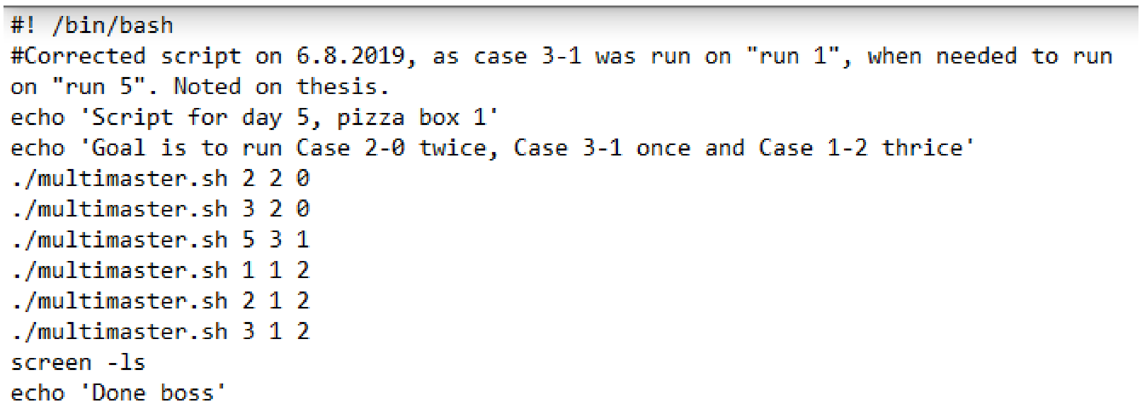


```
#! /bin/bash
t="24h"
./day2_pizza1_sh
##echo 1
sleep $t

./day3_pizza1_sh
##echo 2
sleep $t

./day4_pizza1_sh
##echo 3
sleep $t

./day5_pizza1_sh
##echo 4
sleep $t

./day6_pizza1_sh
##echo 5
sleep $t

./day7_pizza1_sh
##echo 6
sleep $t


./day8_pizza1_sh
##echo 7
sleep $t
```

Comparison of Chapter 3.4.5 Day 5 scripts, repaired script on bottom:



```
#! /bin/bash
echo 'Script for day 5, pizza box 1'
echo 'Goal is to run Case 2-0 twice, Case 3-1 once and Case 1-2 thrice'
./multimaster.sh 2 2 0
./multimaster.sh 3 2 0
./multimaster.sh 1 3 1
./multimaster.sh 1 1 2
./multimaster.sh 2 1 2
./multimaster.sh 3 1 2
screen -ls
echo 'Done boss'
```



```
#! /bin/bash
#Corrected script on 6.8.2019, as case 3-1 was run on "run 1", when needed to run
on "run 5". Noted on thesis.
echo 'Script for day 5, pizza box 1'
echo 'Goal is to run Case 2-0 twice, Case 3-1 once and Case 1-2 thrice'
./multimaster.sh 2 2 0
./multimaster.sh 3 2 0
./multimaster.sh 5 3 1
./multimaster.sh 1 1 2
./multimaster.sh 2 1 2
./multimaster.sh 3 1 2
screen -ls
echo 'Done boss'
```

APPENDIX 2: EXECUTION SPEED MIN /MAX, AVERAGES AND STANDARD DEVIATIONS

Case 2 Native execution speed averages, minimum and maximum values

	r1		r2		r3		r4		r5	
	Min / Max	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**
1m1s	1197,83		1092,36		1079,71		1277,86		1211,50	
M1	456,93 / 2003,14	1342,09 / 406,15	357,73 / 1877,52	1276,19 / 413,58	332,76 / 1923,43	1346,61 / 364,93	434,80 / 1769,33	1382,83 / 278,88	323,98 / 1790,89	1353,86 / 375,68
S1	502,00 / 1769,94	1196,54 / 166,72	506,73 / 1696,46	1090,99 / 205,00	408,64 / 1856,83	1077,75 / 200,90	271,79 / 1855,27	1276,44 / 215,14	386,32 / 1716,72	1210,52 / 189,25
2m	1480,96		1322,35		1368,84		1326,36		1431,50	
M1	285,91 / 1804,69	1506,28 / 157,76	346,09 / 2185,71	1300,62 / 317,86	458,46 / 2137,64	1371,51 / 294,67	401,51 / 2260,21	1370,88 / 274,17	503,23 / 2005,38	1443,12 / 210,31
M2	244,50 / 1793,08	1446,87 / 175,05	531,51 / 2305,66	1327,81 / 215,00	442,05 / 2049,56	1368,18 / 170,17	385,95 / 2208,19	1305,89 / 243,60	435,03 / 2016,30	1409,73 / 247,00

Case 2 Virtual execution speed averages, minimum and maximum values

	r1		r2		r3		r4		r5	
	Min / Max	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**
1m1s	1247,00		1123,12		1290,27		1114,10		1318,65	
M1	295,77 / 2138,20	1577,99 / 428,85	419,38 / 2101,30	1557,27 / 494,82	388,90 / 2019,82	1509,63 / 364,22	293,75 / 1996,55	1496,85 / 400,65	377,34 / 2311,63	1570,92 / 493,30
S1	195,00 / 2064,68	1244,13 / 262,29	365,36 / 2033,32	1120,50 / 207,26	205,88 / 1998,82	1287,76 / 199,15	198,14 / 2102,06	1110,97 / 259,58	253,66 / 2394,07	1316,51 / 219,76
2m	1444,13		1553,47		1587,96		1481,63		1534,24	
M1	386,09 / 2196,89	1468,57 / 260,55	258,94 / 2258,35	1555,36 / 206,18	336,78 / 2241,28	1569,38 / 213,42	396,08 / 2344,50	1477,53 / 230,18	302,62 / 2256,98	1511,96 / 329,31
M2	335,70 / 2241,43	1370,93 / 349,11	299,34 / 2194,45	1523,14 / 352,42	320,00 / 2175,79	1611,63 / 201,61	369,20 / 2271,45	1485,50 / 231,78	252,83 / 2238,55	1540,05 / 244,60

Case 3 Native execution speed averages, minimum and maximum values

	r1		**r2**		**r3**		**r4**		**r5**	
	Min / Max	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**
1m2s	1218,67		1259,43		1162,44		1166,92		1082,78	
M1	318,89 / 1944,84	1353,74 / 330,26	358,85 / 1954,74	1476,50 / 314,93	510,93 / 1775,39	1293,54 / 346,24	370,67 / 2505,13	1409,78 / 345,33	258,99 / 2100,32	1336,60 / 414,75
S1	406,47 / 1871,47	1190,89 / 185,44	376,90 / 1719,99	1253,27 / 192,62	510,71 / 1775,72	1154,22 / 156,73	413,35 / 2519,30	1159,63 / 152,88	259,46 / 2162,05	1090,56 / 168,93
S2	340,85 / 1901,42	1245,38 / 169,99	344,44 / 1745,53	1263,70 / 183,03	469,00 / 1717,89	1169,59 / 143,97	451,85 / 2434,10	1171,40 / 151,33	307,48 / 2186,73	1073,78 / 194,29
2m1s	1196,99		1212,29		1355,99		1013,88		1189,73	
M1	403,89 / 1939,93	1276,28 / 402,45	497,51 / 1793,76	1290,66 / 273,71	540,36 / 1774,19	1404,42 / 91,40	395,75 / 2163,75	1240,69 / 410,06	533,63 / 1852,08	1314,22 / 314,69
M2	279,96 / 1401,47	1042,24 / 268,08	455,02 / 1757,84	1282,07 / 291,30	550,46 / 1791,20	1377,12 / 285,30	398,73 / 2009,39	1251,71 / 362,23	451,00 / 2000,00	1264,00 / 363,50
S2	367,91 / 1885,89	1197,83 / 238,81	437,58 / 1678,37	1209,75 / 172,40	531,72 / 1806,14	1332,53 / 187,57	336,19 / 2018,39	1009,49 / 181,57	495,61 / 1829,36	1186,81 / 191,92
3m	1225,85		1354,69		1374,77		1262,70		1425,26	
M1	456,66 / 1880,28	1219,93 / 201,84	455,32 / 1746,87	1318,73 / 134,80	462,42 / 1763,08	1353,90 / 189,15	471,46 / 1687,78	1244,44 / 140,61	448,86 / 2049,28	1439,07 / 226,36
M2	460,76 / 1932,71	1219,44 / 200,88	524,16 / 1907,16	1385,57 / 113,21	296,03 / 1864,61	1408,89 / 192,84	373,48 / 1725,05	1267,96 / 323,97	501,57 / 2028,37	1460,22 / 262,91
M3	449,34 / 1864,32	1283,72 / 281,53	461,38 / 1805,13	1359,13 / 128,06	410,86 / 1737,72	1375,22 / 183,79	405,87 / 1924,59	1368,43 / 283,39	436,83 / 1979,13	1400,33 / 182,30

Case 3 Virtual execution speed averages, minimum and maximum values

	r1		r2		r3		r4		r5	
	Min / Max	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**
1m2s	1066,99		1345,81		1346,65		1234,71		1266,49	
M1	275,12 / 2085,79	1453,32 / 427,59	306,01 / 2095,53	1474,69 / 414,32	424,98 / 2043,69	1495,26 / 386,27	419,92 / 2257,34	1558,67 / 484,77	258,27 / 2190,03	1515,16 / 464,59
S1	417,46 / 2116,65	1077,37 / 179,81	204,78 / 2051,57	1343,30 / 191,54	318,84 / 2079,23	1360,95 / 187,65	275,72 / 2210,04	1258,97 / 202,87	248,46 / 2206,88	1272,76 / 219,89
S2	402,15 / 2034,76	1053,88 / 166,63	209,76 / 2178,83	1346,73 / 193,26	432,07 / 2016,18	1330,66 / 184,35	367,43 / 2228,06	1207,38 / 212,44	278,23 / 2231,06	1258,49 / 225,29
2m1s	1151,97		1340,72		1282,35		1296,10		1134,50	
M1	323,82 / 2107,82	1351,00 / 449,17	348,39 / 2021,79	1337,51 / 488,61	309,28 / 2094,69	1389,10 / 413,21	440,51 / 2275,50	1367,27 / 395,41	372,13 / 2135,77	1352,16 / 442,26
M2	441,60 / 2121,63	1279,26 / 433,64	322,95 / 2004,03	1420,00 / 423,38	310,69 / 2048,39	1374,19 / 475,41	437,86 / 2303,05	1349,75 / 415,09	307,77 / 2187,72	1434,64 / 503,42
S2	299,05 / 2102,95	1146,85 / 164,05	328,76 / 2073,14	1339,90 / 196,75	373,18 / 2015,47	1280,37 / 199,81	288,94 / 2186,31	1294,23 / 197,76	309,34 / 2164,82	1129,50 / 223,99
3m	1374,91		1335,67		1361,06		1383,30		1258,84	
M1	524,20 / 2084,01	1362,17 / 131,27	371,29 / 2226,67	1287,84 / 233,89	363,67 / 2385,85	1415,03 / 287,72	396,28 / 2149,19	1355,92 / 209,60	387,46 / 2171,06	1252,23 / 168,60
M2	315,09 / 2204,89	1468,85 / 224,59	304,72 / 2005,48	1403,93 / 326,20	357,11 / 2134,77	1315,18 / 377,59	347,09 / 2050,42	1405,16 / 208,87	329,29 / 2094,75	1267,06 / 186,05
M3	558,74 / 2219,50	1343,18 / 168,03	309,26 / 2201,20	1474,43 / 390,32	283,31 / 2361,30	1257,24 / 445,43	313,91 / 2171,61	1392,48 / 210,41	428,95 / 2256,65	1258,63 / 358,17

Case 4 Native execution speed averages, minimum and maximum values

	r1		**r2**		**r3**		**r4**		**r5**	
	Min / Max	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**
1m3s	851,56		1149,71		931,32		1234,26		1082,65	
M1	381,57 / 1864,15	1385,92 / 337,43	328,38 / 1759,36	1399,96 / 333,03	328,72 / 1778,14	1302,47 / 438,90	463,08 / 1867,48	1334,47 / 315,75	378,30 / 1872,73	1326,70 / 398,12
S1	296,35 / 883,54	618,97 / 76,41	369,15 / 1701,54	1146,26 / 166,05	368,01 / 2066,71	940,82 / 174,08	473,03 / 2000,00	1232,92 / 120,43	368,07 / 2048,69	1167,42 / 186,93
S2	534,19 / 1817,49	1303,21 / 160,83	397,91 / 1692,09	1148,75 / 165,84	358,39 / 1996,00	933,80 / 164,92	507,72 / 1748,17	1235,80 / 125,80	222,73 / 1920,00	1032,26 / 167,49
S3	303,26 / 882,72	617,57 / 76,62	278,76 / 1688,79	1152,58 / 163,36	374,06 / 1800,48	917,85 / 165,68	490,52 / 1838,53	1233,08 / 129,36	326,04 / 1855,72	1045,91 / 168,25
2m2s	1029,87		1158,43		931,47		931,47		1160,85	
M1	405,33 / 2000,00	1189,50 / 385,36	385,38 / 1980,78	1204,34 / 406,36	441,19 / 2000,00	1198,60 / 398,09	460,61 / 1778,51	1187,14 / 379,18	395,41 / 1827,68	1327,65 / 403,09
M2	350,02 / 2000,00	1263,07 / 382,91	329,86 / 2063,20	1219,83 / 417,21	451,10 / 1882,83	1222,38 / 349,59	517,34 / 1875,68	1263,94 / 389,23	512,13 / 1932,14	1286,11 / 348,37
S1	425,36 / 2000,00	1040,89 / 176,52	449,13 / 2041,65	1174,03 / 153,49	251,01 / 1458,47	791,86 / 133,14	475,55 / 1756,08	1110,66 / 142,80	471,01 / 1840,71	1253,95 / 148,96
S2	393,88 / 2000,00	1015,55 / 176,02	411,61 / 1944,76	1141,84 / 155,47	312,14 / 2000,00	1064,66 / 167,92	522,17 / 1861,78	1210,14 / 148,31	536,00 / 1850,48	1221,63 / 148,11
3m1s	1011,26		1198,16		1204,12		1222,73		1211,40	
M1	379,05 / 1832,88	1103,86 / 344,87	397,86 / 1804,20	1272,34 / 336,97	339,33 / 2000,00	1301,35 / 351,77	301,98 / 1763,33	1234,21 / 230,48	316,93 / 2000,00	1227,67 / 181,74
M2	319,78 / 1794,81	1197,78 / 360,20	344,22 / 1775,36	1222,91 / 386,85	340,47 / 2000,00	1167,89 / 194,04	480,05 / 1816,46	1213,32 / 306,06	527,28 / 1760,64	1136,09 / 294,93
M3	375,12 / 1891,17	1100,43 / 378,91	376,19 / 1783,57	1145,08 / 159,50	381,79 / 1801,53	1359,62 / 308,47	450,87 / 1734,67	1227,46 / 130,60	484,71 / 1853,21	1250,13 / 122,12
S1	332,89 / 1979,15	1006,51 / 170,66	392,61 / 1784,19	1202,98 / 202,87	304,14 / 1765,63	1202,31 / 194,53	422,38 / 1688,75	1221,72 / 168,03	432,53 / 1854,42	1205,48 / 164,14

Case 4 Native execution speed averages, minimum and maximum values

	r1		**r2**		**r3**		**r4**		**r5**	
	Min / Max	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**
1m3s	1245,17		1098,33		1195,04		946,80		1162,91	
M1	480,96 / 2130,28	1462,74 / 431,44	400,02 / 2042,64	1502,80 / 425,90	297,34 / 2090,11	1500,11 / 431,44	350,58 / 2033,22	1405,46 / 519,79	257,95 / 2063,86	1622,00 / 416,96
S1	348,54 / 2101,06	1239,00 / 159,05	273,92 / 2066,28	1095,13 / 193,50	277,92 / 1966,29	1186,74 / 170,15	296,59 / 1936,33	946,23 / 185,64	207,32 / 2154,23	1158,75 / 217,21
S2	328,34 / 2130,80	1251,78 / 171,55	276,84 / 2021,44	1115,25 / 189,12	242,99 / 2071,11	1208,05 / 179,12	333,71 / 1937,73	948,77 / 190,34	206,87 / 2099,50	1160,54 / 209,07
S3	429,67 / 2125,05	1242,94 / 168,55	275,97 / 2007,17	1080,55 / 185,91	262,63 / 2023,88	1187,87 / 169,71	312,58 / 1953,55	943,24 / 190,88	216,22 / 2103,50	1166,19 / 204,46
2m2s	1142,08		1155,44		1050,69		1050,69		1339,70	
M1	273,46 / 2036,17	1383,11 / 521,07	465,94 / 2156,92	1408,62 / 428,89	214,42 / 1997,19	1338,89 / 486,53	277,29 / 2018,76	1360,20 / 393,11	278,73 / 2200,97	1305,87 / 341,05
M2	176,39 / 1980,42	1329,45 / 542,56	293,11 / 2094,20	1392,55 / 478,06	268,88 / 1969,79	1290,95 / 498,11	374,40 / 1975,08	1396,06 / 403,91	289,32 / 2091,88	1400,76 / 430,23
S1	197,84 / 2025,59	1142,97 / 198,50	394,51 / 2109,82	1167,78 / 186,91	192,31 / 2079,21	1060,60 / 220,71	375,50 / 1947,04	1336,49 / 178,60	189,69 / 2162,24	1350,93 / 202,66
S2	207,79 / 2008,91	1137,76 / 188,75	345,95 / 2033,41	1136,91 / 189,21	169,79 / 2116,81	1035,90 / 218,16	245,28 / 1963,63	1342,04 / 184,42	191,69 / 2180,62	1315,06 / 166,99
3m1s	1141,95		1199,74		1290,76				1117,78	
M1	405,73 / 2083,32	1415,34 / 444,95	321,99 / 1993,72	1326,10 / 427,99	234,32 / 2002,12	1336,89 / 206,75			322,46 / 2126,85	1265,16 / 484,65
M2	384,29 / 2068,24	1328,86 / 461,60	327,37 / 2007,61	1350,78 / 440,20	212,59 / 2125,00	1331,22 / 233,60			319,05 / 2130,45	1372,42 / 489,31
M3	285,32 / 2070,50	1367,98 / 449,44	385,88 / 1982,64	1276,36 / 450,37	198,41 / 2060,37	1287,90 / 446,04			359,59 / 2147,81	1270,12 / 504,23
S1	365,47 / 2083,09	1130,27 / 215,68	270,44 / 2048,19	1194,34 / 215,61	213,04 / 2064,68	1283,73 / 210,45			249,10 / 2124,92	1111,89 / 233,84

Case 5 Native execution speed averages, minimum and maximum values

	r1		r2		r3		r4		r5	
	Min / Max	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**
1m7s	1173,23		1101,57		979,26		1043,15		955,51	
M1	430,28 / 2000,00	1261,31 / 357,71	264,86 / 2002,48	1306,79 / 342,18	250,83 / 1769,51	1254,53 / 458,14	247,89 / 1801,15	1369,49 / 370,97	372,04 / 1873,01	1355,70 / 374,58
S1	481,57 / 2000,00	1217,18 / 125,67	227,05 / 2050,91	1105,90 / 174,86	295,32 / 1705,66	986,16 / 165,67	367,06 / 1792,91	1107,37 / 153,14	348,85 / 1627,06	1085,13 / 144,39
S2	559,97 / 1795,09	1211,18 / 116,42	225,35 / 2000,03	1090,52 / 172,69	256,16 / 1667,77	992,56 / 161,80	251,10 / 1768,42	1105,96 / 157,80	253,87 / 861,45	540,73 / 66,50
S3	592,12 / 2000,00	1221,58 / 121,02	230,39 / 2030,75	1116,75 / 177,24	300,95 / 1670,20	980,24 / 161,28	264,38 / 2000,00	1066,64 / 150,81	364,52 / 2000,00	1155,68 / 143,60
S4	569,20 / 2000,00	1210,05 / 122,90	226,13 / 1939,41	1093,13 / 171,83	302,86 / 1612,20	947,99 / 160,64	297,25 / 2000,00	1073,69 / 147,34	236,05 / 831,53	540,80 / 66,40
S5	535,72 / 1704,49	1213,94 / 121,03	231,88 / 2000,00	1101,16 / 172,21	312,70 / 1650,85	945,91 / 156,53	237,86 / 2000,00	1081,05 / 152,00	382,18 / 1780,10	1158,52 / 148,18
S6	573,32 / 2000,00	1219,43 / 117,81	324,12 / 2044,67	1099,00 / 166,67	279,24 / 1765,21	1020,64 / 166,34	324,41 / 1758,63	1073,91 / 150,97	245,00 / 1694,28	1070,04 / 138,26
S7	381,37 / 1600,00	917,94 / 108,86	230,77 / 2000,00	1102,81 / 168,89	305,98 / 1719,00	980,24 / 166,32	155,00 / 1600,00	790,08 / 146,41	389,67 / 1748,35	1129,30 / 136,09
2m6s	1069,99		963,95		998,03		1052,19		1014,38	
M1	378,48 / 1894,71	1223,72 / 385,94	251,99 / 1773,90	1286,59 / 371,73	356,99 / 1964,15	1154,29 / 380,29	425,39 / 1770,20	1191,86 / 341,68	417,73 / 2056,62	1307,45 / 353,50
M2	362,06 / 1674,81	1223,41 / 387,31	407,69 / 1690,58	1203,78 / 334,84	366,59 / 2000,00	1209,82 / 377,57	412,24 / 2000,00	1176,60 / 386,91	413,41 / 2065,32	1266,46 / 386,09
S1	348,43 / 1753,02	1047,48 / 141,96	315,78 / 1669,39	993,11 / 155,03	351,24 / 1910,36	993,17 / 149,45	260,31 / 1436,88	806,63 / 105,43	506,47 / 2110,43	1214,49 / 155,83
S2	369,57 / 1784,93	1091,14 / 140,96	280,56 / 1350,79	719,13 / 132,15	328,06 / 2033,60	1010,32 / 156,68	405,42 / 1810,88	1091,04 / 125,70	438,16 / 2079,88	1215,57 / 150,51
S3	354,20 / 1745,63	1051,62 / 138,65	353,38 / 1714,71	1027,03 / 158,09	346,63 / 1928,70	1008,27 / 152,49	415,24 / 2000,00	1085,81 / 119,17	255,73 / 1021,14	589,61 / 71,58

S4	332,06 / 1716,45	1049,01 / 138,76	387,47 / 1746,37	992,33 / 160,54	328,71 / 2005,41	1001,55 / 157,41	412,17 / 2000,00	1103,72 / 125,49	503,42 / 2149,43	1247,28 / 152,95
S5	411,99 / 1799,71	1099,37 / 141,13	358,26 / 1712,43	1040,93 / 161,07	305,20 / 2000,00	980,49 / 147,50	406,85 / 2000,00	1110,50 / 120,11	254,81 / 984,18	589,59 / 71,01
S6	369,34 / 1789,81	1078,62 / 139,93	356,80 / 1684,11	1005,16 / 153,08	322,22 / 2040,42	991,38 / 153,09	422,64 / 2000,00	1111,98 / 125,43	405,07 / 2120,68	1220,08 / 158,01
3m5s	1260,55		1055,36		997,59		1213,18		1246,54	
M1	470,52 / 1940,43	1244,45 / 103,09	312,49 / 832,42	584,41 / 122,36	274,54 / 1808,12	1130,61 / 365,83	452,43 / 1730,65	1250,08 / 106,52	432,22 / 2060,90	1200,42 / 170,59
M2	491,01 / 2083,68	1240,46 / 323,01	214,96 / 1754,80	1225,50 / 302,42	357,58 / 1747,37	1125,58 / 363,99	402,77 / 1710,87	1244,65 / 339,55	386,27 / 2067,89	1229,48 / 334,25
M3	491,08 / 1997,66	1266,50 / 91,16	198,18 / 1752,79	1218,16 / 319,56	274,04 / 1770,06	1092,41 / 365,09	447,55 / 1752,75	1254,48 / 103,93	342,68 / 1992,08	1198,00 / 306,27
S1	539,42 / 1978,14	1252,57 / 125,34	442,45 / 1804,35	1215,72 / 164,50	295,12 / 2000,00	785,36 / 126,44	396,36 / 1719,07	1215,81 / 115,59	449,91 / 2067,90	1246,25 / 136,82
S2	475,98 / 2068,01	1244,37 / 126,11	388,68 / 1871,23	1242,48 / 158,06	403,85 / 2000,00	1066,47 / 148,61	433,13 / 1752,03	1230,47 / 114,83	414,93 / 2028,76	1233,84 / 140,40
S3	507,19 / 2124,76	1286,51 / 132,93	152,34 / 831,93	367,83 / 115,32	383,94 / 1751,54	1042,85 / 148,25	491,75 / 1726,08	1207,22 / 113,78	487,52 / 2029,88	1264,24 / 135,75
S4	496,87 / 2075,46	1264,88 / 127,84	403,78 / 1746,69	1195,64 / 155,09	444,65 / 1745,00	1032,38 / 148,68	426,30 / 2000,00	1206,80 / 111,24	512,07 / 2055,51	1245,35 / 134,35
S5	551,50 / 2084,56	1255,52 / 133,25	329,67 / 1824,58	1214,90 / 172,77	450,53 / 2000,00	1055,99 / 146,06	497,51 / 1659,59	1180,31 / 109,22	452,39 / 2046,15	1246,71 / 135,82

Case 5 Virtual execution speed averages, minimum and maximum values

	r1		r2		r3		r4		r5	
	Min / Max	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**	**AAvg. / SDev.**	**Min / Max**
1m7s	1262,53		1022,89		1072,16		1094,23		1114,53	
M1	384,35 / 2009,53	1374,78 / 424,37	275,44 / 1929,81	1397,63 / 472,54	329,45 / 2013,38	1470,06 / 486,03	265,21 / 1933,72	1427,50 / 436,02	380,63 / 1937,79	1443,97 / 455,11
S1	482,86 / 2032,46	1261,42 / 129,83	238,54 / 1974,06	1015,67 / 192,84	332,24 / 1945,23	1075,59 / 155,10	216,13 / 1920,00	1070,53 / 172,73	363,48 / 1881,85	1103,13 / 133,01
S2	520,66 / 2000,00	1261,49 / 126,37	206,28 / 1936,03	1029,86 / 190,11	398,28 / 1932,22	1070,99 / 151,73	247,52 / 1857,00	1083,02 / 164,33	212,87 / 1905,31	1127,37 / 139,43
S3	488,13 / 2032,42	1260,61 / 123,50	207,73 / 1867,20	1021,15 / 192,79	336,49 / 1949,20	1075,35 / 150,02	180,77 / 2000,00	1094,91 / 157,94	500,64 / 1899,75	1126,70 / 133,54
S4	547,13 / 1949,57	1265,98 / 127,42	238,74 / 1809,25	998,51 / 184,32	435,25 / 1920,21	1064,71 / 149,63	180,67 / 1917,95	1104,20 / 163,87	391,82 / 1851,49	1104,24 / 129,12
S5	373,02 / 1948,70	1269,59 / 127,20	171,81 / 1929,92	1023,03 / 189,89	417,25 / 1976,15	1077,66 / 151,93	251,48 / 2000,00	1105,59 / 162,30	366,90 / 1889,84	1119,56 / 132,13
S6	432,62 / 1990,56	1258,80 / 123,19	196,08 / 2000,00	1047,40 / 192,59	421,53 / 1901,37	1063,90 / 146,41	227,97 / 1928,64	1098,64 / 165,92	349,95 / 1881,40	1105,34 / 132,93
S7	479,06 / 2005,99	1258,90 / 126,54	207,69 / 2000,00	1022,87 / 191,00	408,74 / 1943,71	1074,86 / 149,02	231,13 / 1902,45	1100,76 / 165,31	307,67 / 1872,22	1113,42 / 131,99
2m6s	975,57		960,99		1036,36		1187,79		972,49	
M1	405,28 / 1919,92	1385,46 / 381,12	345,99 / 2087,22	1338,97 / 466,19	271,25 / 1873,42	1279,85 / 462,62	424,15 / 1893,83	1275,23 / 346,06	216,86 / 1856,87	1200,38 / 425,97
M2	402,66 / 2002,94	1345,88 / 409,54	270,73 / 1856,72	1399,87 / 458,96	236,34 / 935,25	665,64 / 200,42	478,09 / 1939,06	1334,80 / 365,31	357,65 / 1806,00	1225,38 / 426,63
S1	166,38 / 973,19	564,11 / 80,55	188,03 / 1975,00	975,24 / 210,99	200,52 / 1945,27	1211,12 / 149,04	269,84 / 1903,04	1208,73 / 159,75	185,33 / 1846,53	964,23 / 185,92
S2	205,78 / 1926,29	1173,75 / 161,60	195,12 / 1877,90	954,07 / 212,51	195,12 / 1887,23	1170,99 / 146,62	276,77 / 1878,09	1200,86 / 142,71	133,64 / 1881,74	962,33 / 189,02
S3	268,94 / 1945,84	1169,61 / 158,40	188,34 / 1877,77	971,61 / 206,60	176,67 / 1879,43	1143,85 / 145,77	251,33 / 1908,13	1171,71 / 169,68	176,76 / 1921,14	992,20 / 187,48

S4	255,38 / 1981,42	1181,22 / 164,58	190,21 / 1955,22	950,45 / 212,90	117,60 / 940,97	313,99 / 107,65	410,36 / 1924,72	1215,49 / 156,91	147,06 / 1825,32	974,94 / 188,19
S5	180,89 / 974,77	563,69 / 80,89	188,68 / 1920,00	948,81 / 208,26	201,54 / 1897,19	1188,76 / 141,98	469,63 / 1877,19	1175,64 / 155,30	154,23 / 1798,84	956,37 / 187,99
S6	239,06 / 1967,22	1187,70 / 164,28	196,35 / 1888,16	960,72 / 198,04	201,04 / 1883,95	1161,67 / 142,80	440,64 / 1877,09	1151,62 / 175,23	174,50 / 1842,61	981,18 / 191,86
3m5s	1275,54		1140,78		1106,13		1151,52		954,84	
M1	387,57 / 1917,58	1300,47 / 372,87	247,07 / 1883,87	1248,36 / 488,65	176,80 / 1887,43	1246,93 / 453,03	173,19 / 945,08	508,15 / 224,15	274,60 / 2013,09	1247,86 / 499,31
M2	437,27 / 1921,71	1297,73 / 402,63	254,73 / 1889,43	1216,79 / 449,79	174,12 / 1916,17	1196,28 / 450,63	228,38 / 953,89	614,25 / 213,98	295,54 / 1981,30	1293,77 / 521,09
M3	420,21 / 2020,12	1267,62 / 116,56	245,10 / 1952,47	1312,46 / 372,20	171,47 / 2000,00	1303,45 / 490,66	359,15 / 1950,86	1257,70 / 415,24	292,57 / 2084,01	1264,85 / 543,44
S1	448,11 / 1950,22	1259,46 / 167,31	220,33 / 1930,00	1130,99 / 202,96	213,97 / 2000,00	1114,90 / 177,24	357,44 / 1965,10	1159,08 / 205,56	270,10 / 2048,52	947,45 / 193,10
S2	448,62 / 1923,39	1260,40 / 163,30	208,00 / 1918,88	1139,41 / 198,89	185,48 / 1925,00	1124,66 / 179,16	278,31 / 1914,95	1169,66 / 196,04	263,12 / 2054,09	954,38 / 195,49
S3	332,28 / 1932,23	1276,75 / 156,49	217,39 / 1950,00	1144,43 / 201,75	222,75 / 2042,98	1092,18 / 179,39	236,47 / 1953,02	1142,13 / 204,71	305,51 / 2022,52	944,46 / 188,94
S4	455,56 / 1925,83	1278,15 / 160,91	180,67 / 1921,56	1137,19 / 204,22	206,23 / 1930,58	1096,83 / 186,14	240,00 / 1942,67	1153,69 / 202,53	316,46 / 2038,57	957,06 / 193,58
S5	305,08 / 1972,72	1305,00 / 164,99	203,39 / 1965,35	1147,69 / 200,52	216,98 / 2000,00	1097,99 / 180,22	347,41 / 1935,37	1143,45 / 195,48	228,94 / 2000,00	962,79 / 196,42

APPENDIX 3: CASE 1 RESULTS

Case 1 native execution results.

	min exec	average exec	max exec	median	std deviation	coverage	AFL bugs	UB	read_utmp	print_user
r1	423,43	1452,90	2857,36	1559,84	376,75	0,0217	0	0	-	-
r2	447,10	1464,58	2704,27	1516,33	243,52	0,0217	0	0	-	-
r3	384,20	1419,21	2509,19	1452,89	279,13	0,0217	2	1	40001	-
r4	401,26	1480,50	2394,44	1536,17	235,87	0,0217	0	0	-	-
r5	457,31	1514,77	2348,58	1550,60	167,27	0,0218	1	1	68725	-
r6	572,52	1473,93	2266,79	1519,90	269,93	0,0217	0	0	-	-
r7	603,54	1471,40	2144,43	1540,13	236,36	0,0217	0	0	-	-
r8	328,80	1471,49	2037,11	1501,92	187,29	0,0217	2	1	54501	-
r9	348,24	1117,17	2064,97	1129,87	204,03	0,0218	0	0	-	-
r10	458,67	1549,70	1909,09	1567,70	103,34	0,0217	2	1	51246	-
r11	479,13	1401,11	2117,21	1500,90	304,16	0,0218	4	1	0	71555
r12	538,01	1467,63	2112,23	1510,66	251,43	0,0217	0	0	-	-
r13	574,39	1473,37	1966,56	1500,61	147,10	0,0217	1	1	62850	-
r14	474,95	1423,51	1967,37	1450,11	199,45	0,0217	0	0	-	-
r15	473,50	1483,01	1942,13	1538,36	246,50	0,0217	0	0	-	-
r16	403,74	1484,97	1910,54	1532,60	210,61	0,0217	1	1	73483	-
r17	448,12	1539,72	1894,40	1577,21	177,76	0,0217	0	0	-	-
r18	385,60	1541,34	1943,30	1612,79	258,37	0,0217	1	1	27128	-

Case 1 Virtual execution results

	min exec	average exec	max exec	median	std deviation	coverage	AFL bugs	UB	read_utmp
r1	411,23	1580,70	2277,69	1609,69	237,80	0,0218	2	1	37957
r2	454,58	1555,49	2285,36	1575,53	202,48	0,0219	3	1	34735
r3	424,23	1507,10	2331,19	1653,31	600,01	0,0218	1	1	27648
r4	390,61	1469,91	2247,52	1478,87	355,23	0,0219	1	1	34628
r5	405,25	1551,13	2223,02	1569,66	258,62	0,0219	1	1	65194
r6	322,82	1551,50	2309,54	1564,94	262,21	0,0219	2	1	28892
r7	270,27	1584,23	2411,62	1597,93	285,39	0,0219	3	1	24061
r8	339,82	1525,28	2306,20	1531,88	326,93	0,0218	2	1	48670
r9	314,50	1423,99	2227,27	1364,34	369,41	0,0219	2	1	42528
r10	316,83	1554,87	2456,62	1562,50	227,90	0,0219	3	1	26866
r11	273,33	1592,96	2472,89	1609,04	149,28	0,0219	3	1	61538
r12	306,98	1559,49	2299,97	1563,25	346,32	0,0219	2	1	45781
r13	275,40	1599,78	2545,50	1650,10	519,85	0,0218	2	1	6433
r14	305,08	1530,61	2603,51	1577,03	453,77	0,0218	1	1	10866
r15	337,88	1635,00	2322,40	1631,09	331,03	0,0218	2	1	4555
r16	315,53	1587,34	2491,40	1610,33	284,97	0,0219	1	1	18292
r17	432,73	1491,82	2610,67	1501,63	466,02	0,0218	1	1	7732
r18	317,60	1620,69	2713,83	1632,04	293,20	0,0219	3	1	23417

APPENDIX 4: TABULATED MIN/MAX/AVERAGE VALUES USED FOR COUNTING SPEEDUP

Native data

	Configuration	Min	Max	Average
CAES	1m	0,000895	0,000645	0,000686
	1m1s	0,000422	0,000376	0,000398
	2m	0,00038	0,000339	0,000361
	1m2s	0,000286	0,00025	0,000268
	2m1s	0,000286	0,000243	0,000268
	3m	0,000269	0,000233	0,000249
	1m3s	0,000255	0,000199	0,000222
	2m2s	0,000234	0,000196	0,000214
	3m1s	0,000227	0,000199	0,000208
	1m7s	0,000124	0,000106	0,000115
	2m6s	0,000121	0,000113	0,000117
	3m5s	0,000121	9,95E-05	0,000108
	Configuration	**Min**	**Max**	**Average**
read_utmp	1m	27128	73483	53990,57
	1m1s	1223	4534	2878,5
	2m	1223	39397	13873,4
	1m2s	736	919	834,6
	2m1s	695	27751	6251,6
	3m	800	74577	23542,2
	1m3s	629	962	754
	2m2s	960	22118	6004
	3m1s	645	3251	1275
	1m7s	665	726	683,2
	2m6s	655	1384	1000,6
	3m5s	657	778	700

Virtual Data

	Configuration	**Min**	**Max**	**Average**
CAES	1m	0,000702	0,000612	0,000645
	1m1s	0,000383	0,000346	0,000363
	2m	0,000352	0,000314	0,000331
	1m2s	0,000279	0,000239	0,00025
	2m1s	0,000265	0,000244	0,000252
	3m	0,000265	0,00024	0,000247
	1m3s	0,000236	0,000192	0,000205
	2m2s	0,000212	0,000184	0,000195
	3m1s	0,000199	0,000191	0,000194
	1m7s	0,000117	9,79E-05	0,000109
	2m6s	0,000123	0,000103	0,000116
	3m5s	0,000123	9,76E-05	0,000109
	Configuration	**Min**	**Max**	**Average**
read_utmp	1m	4555	65194	30544,06
	1m1s	1354	5068	762,6
	2m	1354	11186	4937
	1m2s	731	1453	960,6
	2m1s	675	1333	912
	3m	882	27509	8531,4
	1m3s	772	1230	927,8
	2m2s	686	917	776,6
	3m1s	669	1780	1185,5
	1m7s	629	870	734,8
	2m6s	627	856	690,6
	3m5s	628	996	733,8