Evaluating eBPF based tracing for analyzing packers on Android

Final Lab Presentation

Overview

- Motivation
- Background
  - Packers, eBPF and BCC
- Implementation
  - Platform, Analysis Logic
- Evaluation
- Conclusion

Motivation

- Over the years, Android became one of the most widely used operating systems [1]
- Unfortunately, this popularity also draws the attention of malefactors trying to benefit from the large userbase
  - For example: Kaspersky registered in 2020 alone over 5 mio. malicious packages for Android [2]

Motivation

- At this scale, development of Android centered Antivirus technology is a natural consequence [3,4,5]
- As a reaction, malware authors may utilize strategies, e.g. in form of obfuscation (packers), to avoid detection
- Subsequently, the result is an evasion-detection “arms race“ between malware and antivirus

Motivation

- To keep an edge in this competition, it is vital to research new and alternative methods of malware detection
- This gave inspiration for this lab project
- The work evaluates how eBPF based tracing is suited to analyze packers on Android
  - eBPF usually associated with performance measurement [6]

Motivation

- For the evaluation, an exemplary eBPF based analysis system was implemented and tested with samples of packers
- Further, the evaluation contains a comparison of eBPF to strace
  - To convey a more general impression of eBPF’s capabilities

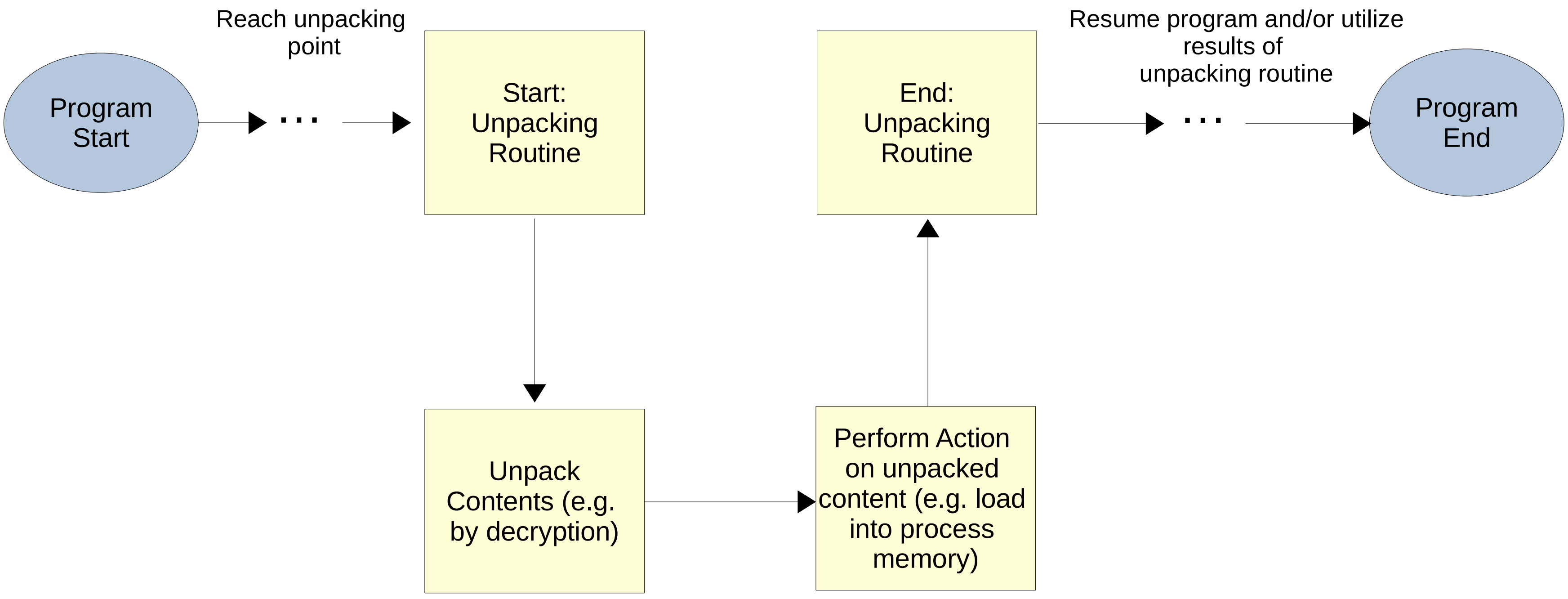
Background

- **Packers**
- eBPF
  - BCC

Background - Packers

- Simplified: a form of obfuscation aiming to protect sensitive content from being directly accessed
- Utilized in malware to avoid detection by antivirus engines by hiding malicious code
- Not only malware related, also used in DRM measures

Background - Packers



Background

- Packers
- **eBPF**
  - BCC

Background – eBPF

- eBPF is short for *extended Berkley Packet Filter*
- Based on BPF, which is a mean to realize high performance packet filtering
- Core component: a virtual machine running within the OS kernel performing these filtering tasks
- Userland programs can load filters (as bytecode) into that VM and have access to the filtering results

Background – eBPF

- eBPF is very similar to the original BPF, with the exception, that the VM is able to access additional event sources
- This enables eBPF to process data not only on packets but also on, e.g. *kprobes* and *uprobes*
  - For example, to trace system and library calls

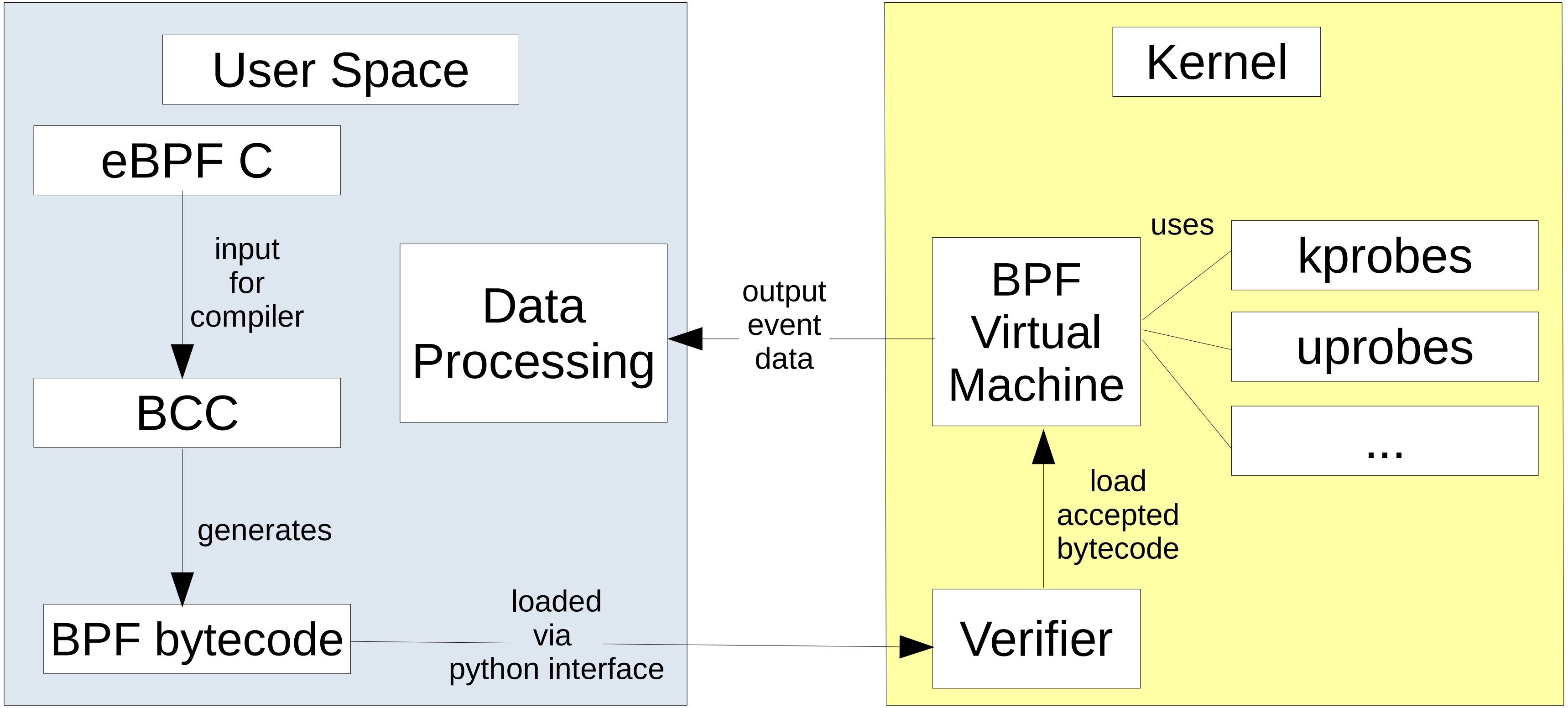
Background

- Packers
- eBPF
  - **BCC**

Background – BCC

- Implementing a meaningful filter in pure BPF VM bytecode is very difficult and error prone
- This is where BCC (BPF Compiler Collection) comes in
- BCC makes working with eBPF much easier as it:
  - Generates bytecode from human-readable eBPF C scripts
  - Loads code into the VM and handles access to the output via convenient python bindings

Background – eBPF and BCC



Implementation

- **Building an Android platform with eBPF support**
  - **Platform Requirements**
  - Platform Selection
  - Installing BCC
- Realizing the Analysis Component
  - Analysis Strategy
  - Analysis Logic

Implementation – Platform Requirements

- In order to support eBPF on Android, the kernel has to be compiled with an appropriate configuration
- Therefore, the target platform should either:
  - already possess an appropriate kernel
  - should make it possible to easily compile and run a customized kernel

Implementation – Platform Requirements

- The platform should be x86-64 based to preserve performance

- Compatibility with as many samples as possible:
  - Running ARM based applications
  - Running older/newer applications
  - Running apps relying on Google APIs

Implementation

- **Building an Android platform with eBPF support**
  - Platform Requirements
  - **Platform Selection**
  - Installing BCC
- Realizing the Analysis Component
  - Analysis Strategy
  - Analysis Logic

Implementation – Platform Selection

- In this work, three potential candidates were chosen:
  - Android Emulator
  - Cuttlefish
  - Android-x86

Implementation – Platform Choice

Name	eBPF out of the box?	Kernel easily configurable?	Compat. with ARM Apps?	Compat. with older Apps?	Google APIs	Performance
Android Emulator	Yes, with Android >=10	Deviations from stock kernel versions difficult	Only with Android 11	Depends on Android version (conflict with ARM compat)	Yes	Good
Cuttlefish	Yes, images with Android >=10	Yes	Only since Android 11	Depends on Android version (conflict with ARM compat)	Yes	Poor
Android x86	No	Yes	Yes, has ARM-Translation	Depends on Android version	Not out of the box	Good

Implementation

- **Building an Android platform with eBPF support**
  - Platform Requirements
  - Platform Selection
  - **Installing BCC**
- Realizing the Analysis Component
  - Analysis Strategy
  - Analysis Logic

Implementation – Installing BCC

- With the suitable platform in place, the last thing to setup was BCC
- A very convenient way to do so is using the open source project *Androdeb (adeb)*
  - Creates a debian based chroot environment that installs all dependencies and automatically builds BCC on the platform

Implementation – Analysis Component

- Building an Android platform with eBPF support
  - Platform Requirements
  - Android Emulator, Cuttlefish, Android-x86
  - Installing BCC
- **Realizing the Analysis Component**
  - **Analysis Strategy**
  - Analysis Logic

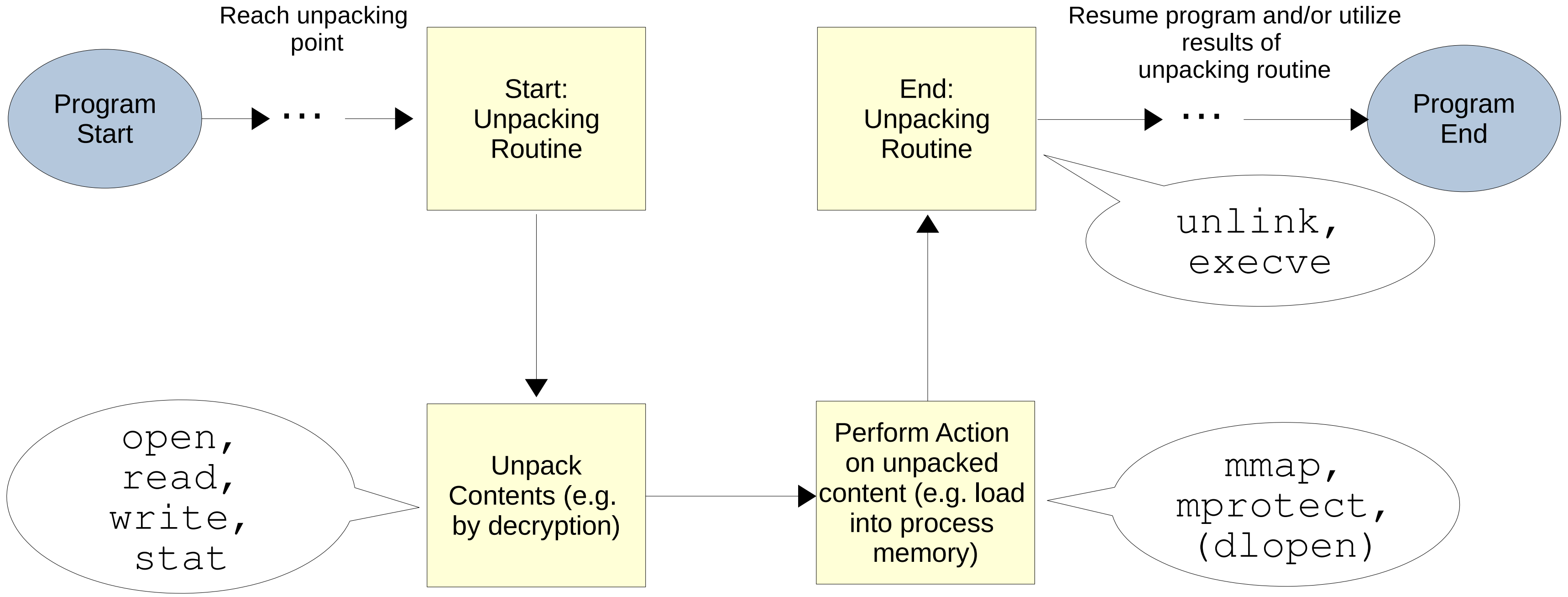
Implementation – Analysis Strategy

- Focus is not labeling apps with *uses/doesn't use packer*
- Rather to showcase if eBPF based tracing can collect sufficient information to, e.g., decide such label
- The inspection strategy should be generally applicable, i.e., no tailoring towards individual applications
- Unpacked contents should be retrieved automatically during the analysis

Implementation – Analysis Strategy

- Assumption for this lab project:
  - Packed malware initially hides harmful code (Dex/ELF) and will, at a certain point, unpack and load this code into process memory
- We know: Programs heavily rely on system calls to perform most tasks
  - So, a viable strategy would be tracing system calls associated with unpacking routines and code loading

Implementation – Analysis Strategy



Implementation – Analysis Strategy

- Limits of this strategy:
  - Methods of unpacking not using any system/library calls
  - For example, code content is written into an executable memory region via a while-loop and direct addressing

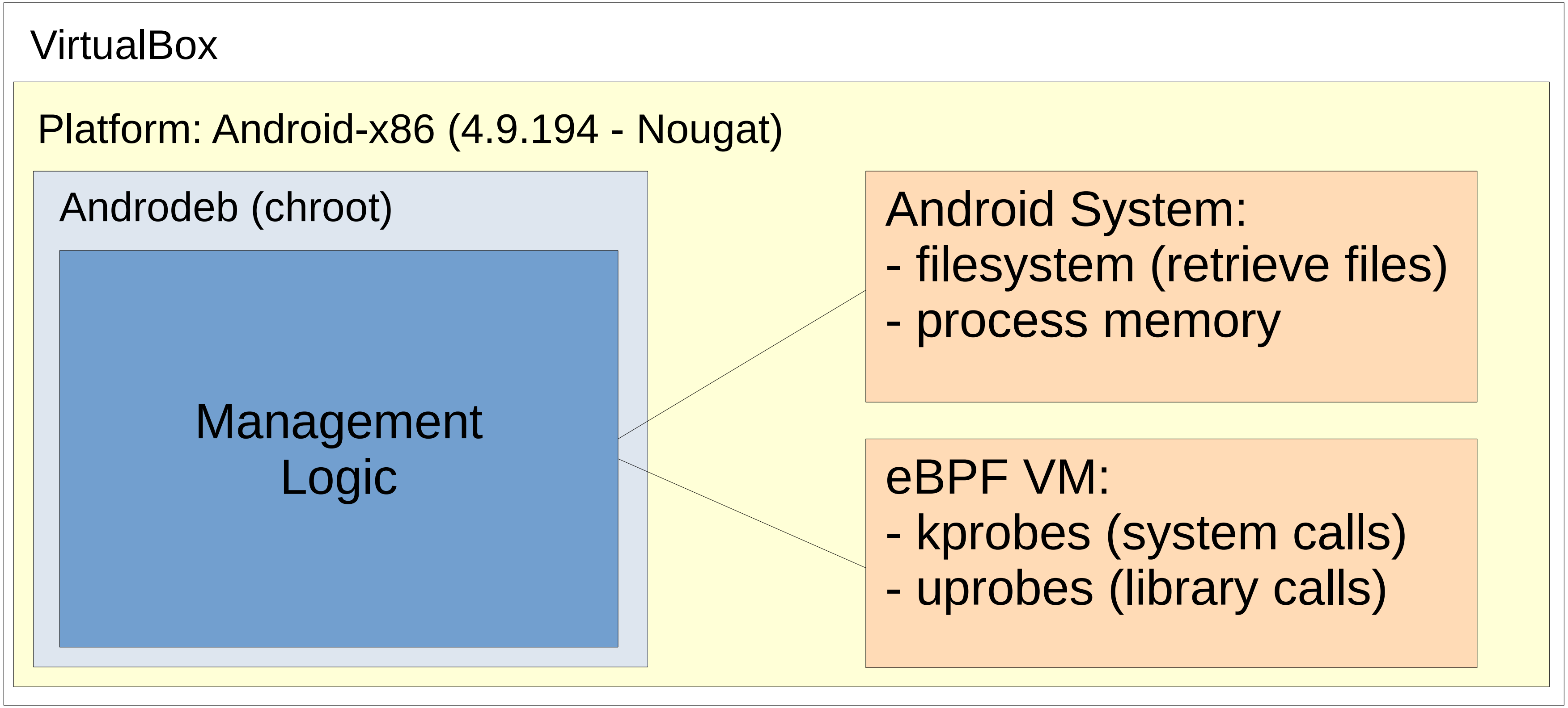
Implementation – Analysis Component

- Building an Android platform with eBPF support
  - Platform Requirements
  - Android Emulator, Cuttlefish, Android-x86
  - Installing BCC
- **Realizing the Analysis Component**
  - Analysis Strategy
  - **Analysis Logic**

Implementation – Analysis Logic

- Analysis Logic consists of two parts:
  - eBPF logic (eBPF C)
    - Logic executed within the VM
    - Retrieves parameters from system calls, library calls, etc.
  - Management logic (BCC python bindings)
    - Compiles eBPF logic and loads the bytecode into the VM
    - Retrieves output from VM, derives additional information and persists the results

Implementation – Analysis Component



Implementation – Remarks

- Problem 1:
  - Direct access to resources, e.g., files within the VM is not possible,
    - Retrieval has to be done via the Management Logic
    - However, this is subject to race conditions
    - This makes the analysis not reliable in that regard
    - Experimental alternative: stitch file contents together from user memory (see next problem)

Implementation – Remarks

- Problem 2:
  - Some information is copied from userpace memory via eBPF
  - This bears the risk that a malicious application can intentionally falsify this information [7]
    - e.g. via another thread that overwrites a memory region eBPF will read / is reading

Implementation – Remarks

- Problem 3:
  - Analyzing 32 bit processes proofed to be problematic:
    - Tracing the 32bit counterpart of system calls often did not yield correct results (parameters were empty)
    - Solution: Find/Trace functions residing “deeper” in the kernel
    - However, portability between kernel versions suffers from this, as kernel symbols can change between versions

Implementation – Remarks

- Problem 4:
  - While rarely being a problem:
    - eBPF can loose trace event data, e.g., if the system is under heavy load (does not process events fast enough) [8]
    - So there is a chance that important data is lost

Evaluation

- **Foreword**
- Analyzing Samples
- Comparison to strace

Evaluation - Foreword

- Evaluation not straightforward, as no objective metric can be given to estimate the aptitude for packer analysis

- To give an impression, the evaluation consists of two parts
  - By using the system to analyze samples of packers
  - By comparing eBPF to strace

Evaluation

- Foreword
- **Analyzing Samples**
- Comparison to strace

Evaluation – Analyzing Samples

- Samples usually retrieved from virusshare.com or koodous.org
- To give an impression here, only samples with a “strong” ground truth were analyzed
  - That is, apps where the unpacked files have already been retrieved and identified
- However, the number of such samples is very small and hard to come by

Evaluation – Analyzing Samples

- For example, a sample app was taken from a Fortinet blog post [9], which showcases how to retrieve a certain unpacked file
  - The author of the post used Frida to manually instrument the process to retrieve the file

Evaluation – Analyzing Samples

- The sample was analyzed with the analysis logic:
  - The traces showed the operations concerning the packed contents
  - Target files were automatically retrieved
    - From file and process memory
  - But race conditions occasionally interfered with results
    - Sometimes more than one run was necessary
- If not for the race conditions, the potential would be there

Evaluation

- Foreword
- Analyzing Samples
- **Comparison to strace**

Evaluation – Comparing to strace

- In this comparison, only the logic running in the VM is considered
- strace is a popular tool, that uses the ptrace syscall to trace system calls of a specific process
- Comparison reasonable, as strace performs a very similar task to eBPF tracing

Evaluation – Comparing to strace

Name	System calls?	Library calls?	Internal kernel funcs?	Detectable by traced program	Filtering	Ease of use	Can loose information?
strace	yes	no	no	simple	Simple filtering	Very simple	no
eBPF	yes	yes	yes	non-trivial	Very complex filters possible	Rather complex setup required	yes, but rarely

Evaluation – Comparing to strace

- eBPF has the ability to tap into much more information than strace
- However, note that strace has all these traces already implemented
  - In eBPF these may have to be implemented beforehand (for VM bytecode)

Conclusion

Conclusion

- eBPF features many very powerful tracing capabilities
  - These can give deep and detailed insights into the system
  - Even more than it is already possible with strace
  - Thus, an understanding about what a packer does can be achieved with eBPF

Conclusion

- However, a meaningful packer analysis also consists of (reliably) retrieving unpacked contents
  - This is where eBPF reaches its limits
  - The eBPF VM does not have direct access to files/resources:
    - Possible solutions:
      - Rely on external routines (race conditions)
      - Stitch together in user memory (falsification possible)

Conclusion

- Even disregarding the retrieval aspect:
  - eBPF comes with its own set of problems:
    - 32bit processes, pot. portability issues, etc.
  - Depending on the number of events, a vast amount of time has to be invested to implement the traces
- Considering this, developing a kernel module is a very viable alternative at this point
  - Modules are also more powerful

Conclusion

- Keep in mind, this work only gives an impression on how eBPF fares in context of the used analysis strategy and how it compares to strace
- There may be other strategies, where the use of eBPF is more beneficial

Conclusion – Future Work

- Using USDT (User statically defined tracing) to get insights into the Java parts of applications
- Experimenting with eBPF’s capability to read/modify user memory:
  - Placing trampolines for hooks
  - Is there a possibility to retrieve file contents reliably?

Thank you for your attention

- Any questions?

If you are interested...

- ... you can find the project including guides and further information under:

https://github.com/JagwarWest/lab-ebpf-based-tracing-for-packer-analysis-public

References

[1] https://gs.statcounter.com/os-market-share

[2] https://securelist.com/mobile-malware-evolution-2020/101029/

[3] https://www.kaspersky.de/android-security

[4] https://www.trendmicro.com/de_de/forHome/products/mobile-security.html

[5] https://www.avira.com/de/free-antivirus-android

[6] https://www.brendangregg.com/ebpf.html

[7] https://github.com/nccgroup/ebpf/blob/master/talks/Kernel_Tracing_With_eBPF-35C3.pdf

[8] https://android.googlesource.com/platform/external/bcc/+/HEAD/docs/reference_guide.md#2_open_perf_buffer

[9] https://www.fortinet.com/blog/threat-research/defeating-an-android-packer-with-frida