Senior Design Project
PlayPen

High-Level Design Report

Ali Altaf Salemwala, Mehmet Can Altuntaş, Usama Saqib
Supervisor: Özgür Ulusoy
Jury Members: Uğur Güdükbay and İbrahim Körpeoğlu

Analysis Report
December 31, 2018

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.
# Contents

1. Introduction  3
   1.1 Purpose of the System  3
   1.2 Design Goals  3
   1.3 Definitions, Acronyms, and Abbreviations  4
   1.4 Overview  4

2. Current Software Architecture  5
   2.1 Market Systems  5
      2.1.1 Sandboxie  5
      2.1.2 Cuckoo  5
   2.2 PlayPen System  5

3. Proposed Software Architecture  6
   3.1 Overview  6
   3.2 Subsystem Decomposition  6
   3.3 Hardware/Software Mapping  6
   3.4 Persistent Data Management  7
   3.5 Access Control and Security  7
   3.6 Global Software Control  7
   3.7 Boundary Conditions  7

4. Subsystem Services  8

5. References  11
1. Introduction

A malware is any software designed to cause intentional harm to a computer or a computer system. This damage is done after it is implanted in to the victim system. Computer Economics published a report in 2007, which estimated the financial damage caused by malware in 2004 to be around 17.5 billion dollars annually [1]. There are millions of malwares in the wild, and more are written everyday. Malware analysis is critical for anybody having to respond to cyber security incidents in today’s environment.

Conventional antivirus systems use signature-based techniques to detect malicious software. This involves maintaining a huge database of known malware and their code-based signatures, and comparing each new file to entries in this database. Unfortunately, this has a high rate of false negatives, and is quite trivial to fool. The sheer number of new malware developed daily means no database can keep up with the pace of innovation, and simple evasion tactics can easily camouflage a known malware from a traditional antivirus system.

Polymorphic viruses, for example, encrypt the malicious portion of their code, thus preventing signature-based detection [2]. Metamorphic virus generators create viruses with new signatures but identical functionalities using techniques such as dead code insertion, code permutation, equivalent code insertion, among others [3]. An antivirus which knows the “ancestor” virus file’s signature is helpless to discover its “metamorphic family” [4].

Furthermore, an essential requirement for a security specialist dealing with a malicious infestation is to gain insight into how the malware behaves, so that rule sets may be generated and supplied to the network security apparatus to flag the relevant behaviour as malicious. Conventional antivirus provide no such insight.

1.1 Purpose of the System

PlayPen intends to automatically and intelligently analyse and detect malwares before they can enter a system by overcoming the limitations of traditional antivirus software and thwarting evasion techniques employed by modern-day malware. It is meant for use by security professionals and enthusiasts, who can use it both as a preventative software barrier between the organization’s system network and external files, and as a diagnostic tool to analyse any malicious files discovered within the network.

1.2 Design Goals

**Administration Cost:** PlayPen intends to reduce the overhead of keeping malware and security specialists on staff. It should thus reduce the work of a system administrator and the security team, with the exception of the additional hardware required.

**Extensibility:** PlayPen is meant to be extendable, as a result the architecture should have interfaces which allow external modules to be added to the system.

**Modularity:** PlayPen should have a modular architecture, where each functionality is enclosed within a module so that it operates independent of other modules. Different functionalities should not interrupt each other. There should be low coupling between modules, and high cohesion between them.

**Performance:** PlayPen should operate as close to real-time as possible, since it will often be analysing files downloaded into the network. These files may be downloaded for immediate use, and should be made available or disposed of as quickly as possible so that the user’s time is not wasted.

**Reliability:** Since PlayPen is a barrier of security for an organization’s computer and network systems, it should be reliable in its performance and results. It should not break down, perform unexpected actions, and should never allow files to pass through without being checked. The results should come from a thorough analysis.

**Robustness:** PlayPen should be a robust system which should work with minimal user interaction or setup. It should not be a delicate software and should have multiple built-in safeguards to prevent any significant crash.
**Scalability:** PlayPen might be used in personal computers or in large organizations. It should scale to use accordingly, and perform equally well in both conditions, where it may have hundreds of requests in a day.

**Security:** PlayPen itself is a security software, and thus should maintain a strong level of security, preventing any unwanted intrusions, releasing of untested files, and access to its analysis modules.

**Usability:** Since the system is meant to be used by laymen as well as experts, it’s client-facing and input-output modules should be highly usable by a non-technical person.

### 1.3 Definitions, Acronyms, and Abbreviations

The following are in alphabetical order, not with respect to significance.

**Control Flow Graph:** A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution.

**Dynamic Analysis:** The analysis of a running file in an attempt to detect suspicious or malicious behaviours.

**Hypervisor:** A hypervisor or virtual machine manager (VMM) is computer software, firmware or hardware that creates and runs virtual machines. They are divided into two types; native or bare-metal hypervisors, and host based hypervisors.

**IOCs:** Indicators of compromise; artifacts or pieces of data that, when found on a network or computer system, indicate with high confidence a computer intrusion.

**Metamorphic Virus Generators:** Generators which construct semantically unique yet functionally stable copies of the original malware.

**Phylogeny:** The so-called “family tree” of a malware showing its evolutionary relationship to other malware.

**Polymorphic Viruses:** A virus with two distinct modules; the first module contains encrypted malicious code; the second decrypts and executes the first.

**Sandbox:** A space on the main computer where programs are run with separated resources, to prevent the spread of failures or exploitation of vulnerabilities.

**Static Analysis:** The analysis of non-running binary file in an attempt to detect suspicious or malicious properties.

**Virtual Machine:** An emulated computer system which runs on a real computer; allows the emulation of different architectures or operating systems.

**Zero-day:** A computer software vulnerability unknown to a party that would be interested in mitigating the vulnerability.

### 1.4 Overview

PlayPen utilizes both static and dynamic malware analysis techniques to classify files fed into it and alert administrators if anything suspicious is found. Both analysis types are used to generate sets of IOCs, which are used for in-depth analysis, detailed below.

The machine learning models, specifically kernel learning, use the IOCs for classification purposes [5]. Phylogeny detection uses Hidden Markov models and opcode analysis to detect whether the current file uses any known malicious techniques [6] [7]. This feature give special insight into the behaviour of the potential malware, by comparing it to known instances.
Since the system is supposed to be automatic, it faces the issue of modern malwares intelligently avoiding detection. For example, most modern malwares operate different code-paths if they discover they are being run within a sandbox. Others randomly obfuscate their code, have null instructions between malicious code, pause at run-time to avoid instant detection, or use a variety of other clever techniques which depend on certain user interactions or particular system conditions [3]. These techniques can be very difficult to preempt, and provide the real challenge in developing our system.

This report discusses the current and proposed software architecture for our project, with the latter providing details about subsystems, hardware/software mapping, data storage, and access control. It then analyses the subsystems and their interactions in detail, providing a high-level view of the whole project.

2. Current Software Architecture

2.1 Market Systems

There are two products on the market which match parts of what PlayPen intends to do. They possess some characteristics and design goals which match ours, and so are included below.

2.1.1 Sandboxie

This ia a simple, lightweight sandbox which allows a user to run programs inside it without having to worry about malicious or harmful software. It does not provide any analysis nor does it return details about the actions of a program. It is intended for casual users hoping to keep their computer safe, not security professionals. Regardless, its lightweight nature and sandbox operations are similar to what we hope to achieve with PlayPen and it is useful as an example of an implemented system. [8]

2.1.2 Cuckoo

Cuckoo is an extensive open-source project which implements some features similar to our own. Cuckoo runs supposed malware in virtualized environments and returns logs of API calls and network traffic, but it does not offer analysis of the results and does not generate any IOCs. It is an example of a system similar to the first half of our project; it should be noted that there are major differences between Cuckoo and PlayPen, however. For example, the virtualized environments in Cuckoo have a much higher level of functionality, while analysis and machine learning services are not provided. [9]

Since Cuckoo is open-source, we can use it’s architecture details as inspiration for the low-level design of our project. The modular nature of the program is a reason we chose a modular design as well, since additional features and functionalities can be added with relative ease. Their development will also not affect the main program, similar to how user-added functionalities to Cuckoo still allow other parts of the program to operate.

2.2 PlayPen System

The previous report was followed by a design revision, and as a result the current architecture of PlayPen is not similar to what it previously was. The revision is not major, however, and as a result the new design still contains many elements of the original design.

The dynamic and static analyses components are essential to the system. Currently, they are separate modules which operate independently and provide distinct results. Both are always run and their results are analysed to provide a complete picture of the provided file. Current designs intend to have users involvement limited to the start and end of the program-flow, meaning users supply a file and then view the results, without interacting with the system midway through, thus limiting potential problems with the analyses as a result of contamination or poor
methodology choices. An exception to this is having the user choose whether they want to run static or dynamic analysis, or both, thus giving them the freedom to view the kind of result they are interested in.

3. Proposed Software Architecture

The newest design is an improvement on the old one, listed in the previous section. There are a few changes, but the overall system and its goal are not changed. The changes and their motivations are discussed below.

3.1 Overview

The software retains its modular design, with an added structure allowing the integration of further modules along the road. This will allow user and developers to easily include more analysis techniques specific to their requirements. However, since there are no current plans for the inclusion of further techniques, this design is only kept in mind for the architecture as a whole, and is used for building the current feature set. It is not explicitly designed for future possible modules. This explicit intention of modularity with a well-defined input-output system has the added benefit of ensuring PlayPen retains its original goal as a malware analysis tool, preventing it from being side-tracked during development into giving prevalence to other features or functionalities.

The new design has a straightforward user flow, where a file is first provided as input, preliminary analysis is used to determine how it is to be handled, static analysis is run to determine potentially malicious elements, and based on both these results and previous user configurations, dynamic analysis is conducted, with the results of both analyses investigated by a machine learning model, and the final classification and further information logs given to the user as output.

3.2 Subsystem Decomposition

Each element of the program-flow described in the previous section is further subdivided into modules which can work either sequentially or in parallel. The modules themselves are subsystems of the entire PlayPen system, and are in turn composed of further subsystems. The difference between the two layers is that the module layer is relatively strict in its structure and program flow, whereas the sub-module layer is highly flexible and customizable, with a program-flow adaptable to user needs.

Figure 1 shows the overall subsystem model at the module level. The placement of the subsystems also provides an idea of the general program-flow.

![Figure 1. Subsystems of PlayPen](image)

3.3 Hardware/Software Mapping

Hardware requirements are not intense, but are also not negligible. The dynamic analysis and machine learning components both require time and computational power to work till conclusion. The client-side of the software is very light, since it only needs to upload a file and display a result, at the most storing a log or quarantining a file. The server-side, however, has to run all analyses and needs a powerful computer to not only run them, but also handle multiple requests which will undoubtedly increase as the organization grow. Here the administrator can set up a system with the requisite power, and upgrade it over time. A system which can support highly threaded operations is enough, so that it can run the many analyses sections in conjunction.
3.4 Persistent Data Management

The primary data which needs to be stored is the statistical data used for the machine learning analysis. This data is essential to the system and will be stored in many locations with many backups, as any loss here will hamstring the entire project. Since most of the data consists of statistical models, the storage system does not need to be as complex as it needs to be robust, and a result a basic database system and simple file hierarchy are enough for this project.

Static and dynamic analysis both create data when they are run, but they do not require any data to be executed. As a result, the information produced will be stored locally in the form of logs, in case the system administrator wants to check the results in more detail or at a later date. Apart from this, the data created here is not essential to the running of the system and does not need a robust system of storage.

3.5 Access Control and Security

Access to the software and its data is tiered. This is essential given the fact that the project itself deals with matters of computer security. Settings and configurations are accessible only by administrators, with special passwords and a new interface, whereas normal users will be able to access simple upload interfaces where they can provide a file for analysis. The results are viewable by all, while administrators can set access restrictions to detailed logs if they desire.

The data and programs used for creating the modelling and analysis sections of the project are highly restricted, as a compromise of these sections compromise the entire system, especially given the centralized nature of some parts of the architecture. These items are kept away from any local sites or client-side programs, and access to them via code is restricted to approved sections of the system, and is not provided via any public API. Human access requires a pre-existing authorized account and authentication.

3.6 Global Software Control

The PlayPen software is event-driven. A file upload or download triggers an analysis, where each step at the module layer depends on the results of the previous step. At the submodule layer, it can be sequential or parallel. The client side provides control to services at the server-side, which returns results triggering the client output.

3.7 Boundary Conditions

Initialization: When initialized, PlayPen readies itself to intercept any files which attempt to enter the network. It also readies an upload screen where users can submit files to be analysed. Other services do not require a startup, nor are they involved in the initial processes, so they are not initialised and are only used after a file is intercepted or uploaded.

Termination: Upon termination, the system will save all current logs and display a shutdown message. If an analysis is in progress, it will ask permission to continue, and if denied will safely cease all actions before storing any incomplete logs. It will quarantine and store the file until it is analysed again or deleted. After all data is cleared and paths are closed, it will shut down. If a shutdown is forced, it will first delete the file and close any running sandbox or hypervisor. After this it will shut down after simply recording a forced shutdown.

Failure: In the event of failure, the system will not have been able to analyse or classify the file for any reason. The system will output details such as the file, its origin, the time, the point of failure, and any deduced reason for the failure to a special failure log. It will notify the user that analysis failed and will not permit access to the file, while especially notifying the administrator of the failure to allow further analysis and action if needed. The file which caused the failure will be quarantined and stored if possible until the administrator chooses to remove it.
4. Subsystem Services

Figure 2 shows a decomposition of the Decider module. This subsystem uses preliminary analysis and configuration files to decide the analyses to be conducted.

The Entropy Analysis component is used to figure out if we are dealing with a polymorphic virus, i.e. an encrypted, packed, or otherwise obfuscated file that needs to have a “reversal” operation performed on it. If this is the case we have no option but to conduct only dynamic analysis since the actual code of the file is obscured.

The Configuration File component is there to give the user flexibility in determining which sorts of analysis to conduct. For example a system administrator may determine that dynamic analysis is too resource intensive and therefore only static analysis must be conducted. If the system detects an encrypted file it is to be automatically blacklisted, without conducting dynamic analysis, in this case.

![Figure 2. Decider Module](image)

Figures 3 shows the Static Analysis module. This subsystem collects data from the plain binary file. It contains three components, each providing a separate source of data.

The Disassembler disassembles the binary file to yield its assembly instructions. These are used in classification and phylogeny detection.

The Control Flow Graph Generator takes the assembly instructions and constructs a control flow graph of the binary. This is also used in both classification and phylogeny detection.

The PE File Information Scraper constructs a feature vector of seven categories of information about the binary file: Entropy, Binary Size, Packed, Number of Vertices (CFG), Number of Edges (CFG), Number of Static Instructions, and Number of Dynamic Instructions. This information is used only in classification.
Figure 3. Static Analysis Module

Figure 4 show the components of the Dynamic Analysis module. This subsystem collects data by tracing the execution of the file.

The Hypervisor Generator component constructs a hypervisor as a safe environment for executing the malware using the Ether framework, because this framework allows the tracing of a file’s execution via a trapping function which logs every CPU instruction [10]. The hypervisor may be generated only once, or with every analyses. This can be set up by the administrator.

The Execution Tracer component parses the logged trace and outputs only the instructions, stripping away all other data. For example given \texttt{mov esi, [ebp+8]} as an x86 instruction, only \texttt{mov} is parsed while the rest is stripped and discarded. This information is used in both classification and phylogeny detection.

The System Call Tracer component parses the trace and collects all the system calls issued by the executing file. The operating system and malware interaction is quiet cleanly parameterized, since malwares generally require a small and distinct subset of system calls in order to function. This information is used in classification.

Figure 4. Dynamic Analysis Module
Figure 5 shows the *Malware Analysis* module, which uses results from the previous modules to classify the file and conduct phylogeny analysis if a malware is detected.

The *Markov Chain Generator* collects the instruction traces, disassembled instructions, and system call traces into separate Markov Chains, which are the primary data representation used.

*Similarity Matrix Generator* uses the Markov Chains from the previous component to compute the kernels of the different data sources. Using the algebra below, we combine kernels of different data sources and use them all in classification:

\[
K_3 = K_1 + K_2
\]

Given two valid kernels, \( K_1 \) and \( K_2 \), \( K_3 \) is also a valid kernel if \( K_3 = K_1 + K_2 \) \[11\]

The *Multiple Kernel Learning* component is where the classification is actually performed. So far our approach involves using Support Vector Machines to perform the actual classification \[12\].

The *Phylogeny Detection* component is used to determine if the malware analyzed is an instance of a metamorphic family of viruses. It has been shown that Hidden Markov Models are very well suited for this kind of detection \[13\]. However, there are some known statistical attacks on Hidden Markov Models, and in order to buttress our system against these, we also employ opcode analysis techniques. This technique uses a similarity function between two Markov Chains to conclude if one is the metamorphic instance of the other \[8\].

Malware authors are also prone to reuse known techniques from previous malwares. Therefore if we can identify which technique has been reused, we can save the analyst a significant amount of time by taking away the need to manually reverse engineer the malware. To accomplish this, we intend to try and find subgraph isomorphisms between the Control Flow graphs of the input file and known malware. Given that subgraph isomorphism is terribly resource intensive, this idea still needs further development. However, previous research suggests that the problem can be made tractable by using intermediary languages \[13\] to construct Annotated Control Flow graphs, reducing the size of the original graph and thus the problem \[14\].

Two additional input and output modules handle boundary cases, but they have no further decompositions and are not shown.
5. References


