Alexandru Ioan Cuza University of Iasi

Faculty of Computer Science

PhD Thesis

Dynamic analysis of possible malware applications

Summary

Supervised by:
prof. dr. Dorel Lucanu

PhD Student:
Vlad Constantin Craciun

Iași, 2021

Abstract

The fast evolving security technologies in the last two decades enabled threat actors to improve their skills and the way they operate, such that cyber security companies had to adapt by developing and updating old architectures. While some cutting edge cyber-security companies still choose to design in-house tools to speedup specific tasks, others prefer to buy or use open source software. The landscape for binary threats is at a point where manual intervention is required only in delicate and sensitive scenarios. However, in practice, manual analysis and debugging may be used more than we expect at the moment. The choice for manual analysis relies mostly on the lack of technologies to automatically handle obfuscation, packing and analysis-evasion techniques.

In this thesis, we propose a series of specific tools for binary analysis, which are further used to design two more complex projects: CFGDump (memory dump reconstruction for running processes based on inter-procedural control flow graph differences) and COBAI (Complex Orchestrator for Binary Analysis and Instrumentation). While CFGDump aims to assist reverse engineers with fast unpacking polymorphic binaries into binaries ready for static analysis (using three of the stand-alone technologies), COBAI has been developed as a full analysis environment based on a DBI engine written from scratch with a dynamic architecture. Unlike its predecessors, COBAI is not meant to be a framework (but it provides a minimum SDK to build additional plugins) and comes equipped with an arsenal of technologies ready to face most up-to-date binary threats in an out-of-the-box manner, ignoring the programming skills of those operating it.

Contents

1. Introduction

During the last decade, the cyber threat industry has been flourishing, taking advantage of:

- cryptography (mostly used in ransomware[1] threats);
- anonymization of payments (using blockchain infrastructure to pay ransoms);
- dark web and Tor project (to host RaaS[2](Ransomware as a Service) pages);
- exploitation of the open source community (using apparently benign applications with compromising purpose);
- unsynchronized fast evolving software and their corresponding old hardware (discontinued Operating Systems would not receive updates anymore);
- lack of knowledge among individuals and companies getting exposed to security gaps when their not-mature enough OS and network infrastructure met the online outer world.

While a few years ago the threat landscape allowed threat analysts to manually reverse engineer payloads, scripts and binaries, the growing of blockchain and the dark web levereged the variety and complexity of threats such that today it is more common for an analyst to use an automated analysis environment, or a set of tools to reverse engineer a threat, spending less time with execution, and more time with filtering the results.

Automating the reverse-engineering of malicious threats can significantly move the analysis time from a few days/hours to a few seconds. However, while tech-

[1] https://www.kaspersky.com/resource-center/threats/ransomware
[2] https://www.crowdstrike.com/cybersecurity-101/ransomware/ransomware-as-a-service-raas/

nologies (emulation, binary-translation, virtualization) and tools to fast forward the analysis already exist, threat authors pushed analysis evasion mechanisms into their work, being able to avoid/deter execution when certain conditions are met. Trying to hide the analysis environment and tools while digging for the root behaviors in a timeless fashion is not easy to achieve.

While most threat analysts are comfortable with multiple tools and programming languages, the problem itself is not different than profiling a benign piece of code. The difference reside in the obfuscation layers present inside the threat, such that usual profiling tools will fail.

There is still a gap between the academic approach (focusing more on unification and soundness) and the industry approach mandating for performance and caring less about all the other details. As seen until now, most companies are more focused on performance rather than generic and abstract approaches. The difference in this case is given by the lack of data in the academic field, and the unwilling of companies to open their data and projects. To fill this gap, academic solutions should concentrate some efforts into achieving performance (with a penalty for soundness and unification), and security companies should open at least some trivial data sets to the research community. An ideal model for binary analysis must include:

- **transparency** - for the analysis tool/environment as well for the entire operating system; while existing transparency policies are mostly static (the policies are adjusted by hand, when issues are found), we believe that dynamic/auto-adjusting ones can be designed;
- **consistency** - to not produce unexpected results and if possible trigger malicious payloads when by default they should not;
- **relevance** - a ranging frame between fine-grained and coarse levels, such that a single analysis would provide at least the desired results;
- **performance** - where possible, analysis should perform faster than the native speed (by run-time filtering the garbage code, and optimizing the execution flow and delays), and where not possible, a slow performance should be

balanced by a comprehensive output;

- **correctness/soundness** - given a large and various data-set, solutions should provide a high rate of accuracy, not crashing or executing unexpected behaviors;

This work proposes a couple of particular solutions to facilitate the analysis of binary threats, along with some larger binary manipulation tools integrating these particular solutions. Currently we focus on the first four items described above, as proving soundness would require a more elaborate mathematical model, and in lack of such a model we consider a wide range of input. Our vision is derived from industry oriented scenarios, where an analyst works much faster changing the configuration, rather than writing a piece of code to conduct the analysis, helped by a framework. We believe a framework is useful at assisting into proving a theory, but does not scale for large inputs. In scenarios with a large input, one millisecond of delay gets amplified to hours and days.

Our proposed solutions involves binary translation and control flow graph manipulation, with additional ingredients to fit many the problem smaller pieces (corner cases and sub-problems) into a clear picture. The solutions are based on more than ten years of threat analysis and research, involving existing tools, frameworks and customized approaches. Our solution should assist threat analysts with relevant content for their tasks, reducing the effort and operational complexity, as well as building auto-analysis services. Adjacently to the research, it is possible for developers to extend the profiling of their projects by optimizing and improving the source-code, operating mostly on the compiled binaries.

Among the practical applications of our research, analysts should be able to extend this work, building centralized (server-client) analysis environments and automate the build of recovery tools based on the analysis of decompiled execution traces.

1.1 Motivation

While existing tools and frameworks (be them commercial or open source) attempt to solve some security related problems, they present some limitations:

1. The range of problems they handle is fixed, such that covering some corner cases, or asking developers to add some features may not always work, or be easy to achieve.
2. Working with frameworks might require programming skills, and by the time a piece of code is designed to solve a problem with that framework, analysts may conclude that to scale it up, the framework itself is not a solution anymore for performance reasons.
3. The lack of input data and test-cases for most open-source projects renders them useless for recent threat binaries. In some cases this lack of test-cases generates crashes or leaves the problem in a floating state, while in other cases the developers only partially handle the problem, lacking the larger picture.
4. The fast evolving technologies, force static behaviors and policies to become dynamic and easy to automate.

To provide some concrete examples for the previous statements, we briefly iterate a few ideas:

1. While IDAPro is known for being a widely used disassembler and a debugger at the same time, it is very unlikely for it to become a non-interactive/automated binary analysis tool in the near future without further efforts. Also while it is relatively expensive (400$ to 4000$ a license) and includes a debugger, some of the reverse engineers up to this day prefer some lightweight debuggers like OllyDbg[3] or x64dbg[4] (as most of the online binary threat analyses include screenshots of their user interface).
2. Angr is a framework for binary manipulation, able to extract control flow graphs among other data. However, the control flow graph extraction for large applications takes from our tests a considerable amount of time, not easy to

[3] https://www.ollydbg.de/
[4] https://github.com/x64dbg/x64dbg

optimize without the developers intervention.

3. PYIATR, PD32 and SCYLLA are open-source tools used to dump and rebuild the import table for process dumped binaries. From our experiments we have seen that PYIATR most of the times generates random bytes inside the import table (making the output unusable). Some similar issues were found on PD32 where for a process with multiple hidden modules part of the same application, it rebuilt standalone modules with individual import tables, instead of building a single module with merged content from all available modules (non-libraries).
4. Given the fact that most of the AntiVirus engines are built to handle previous threats and possible new ones, the fast changing path of malicious actions, also force security vendors to shift architectures and come up with new technologies. Nowadays it is more important to validate the security of an entire network exposed to the online world, rather than having installed security solutions for each system inside the network.

While common Sandboxes (Cuckoo in [11], AnyRun[5], VMRay[6], Falcon[7], Cape[8], Valkire[9], Hybrid-Analysis[10], JoeSandbox[11], VirusTotal[12], WildFire[13], etc.) are able to provide mostly a coarse level of details (relations between parent and child processes, what specific chains of API calls can reveal, what messages are crossing the network, etc.), the fine-grained level (including instructions) require a more fine surveillance of the target application. This fine-grained level can be achieved with a DBI, an Emulator, or a controlled hypervisor. Because an appli-

[5]https://any.run/
[6]https://www.vmray.com/
[7]https://go.crowdstrike.com/try-falcon-prevent.html
[8]https://capesandbox.com/
[9]https://valkyrie.comodo.com/
[10]https://www.hybrid-analysis.com/
[11]https://www.joesandbox.com/
[12]https://www.virustotal.com
[13]https://www.paloaltonetworks.com/products/secure-the-network/wildfire

cation emulator (libemu[14], bdshemu in bddisasm[15], PokasEmu[16]) is limited with respect to the implementation of emulated API functions and certain instructions, and a hypervisor makes efforts to isolate the context of a target process from the rest of the Operating System, a DBI is able to take full control over the analyzed target only, providing hooking mechanisms outside the conventional usage. A statement which we also agree is that the existing DBI engines were not intentionally built for malware analysis (a conclusion derived by AVAST in [10][17], Kirsch Julian et al. in [12] and Daniele Cono D'Elia et al. in [7][18]), a reason for which PIN in [1] and DynamoRIO in [3] are either not able to keep up with threat actors challenges, or the performance lowers considerably without specific optimizations. For instance, for handling code injections and process creation from an instrumented piece of code, DynamoRIO can follow standard process creation (but not the most recent code injection techniques), while using PIN becomes a pain. Other issues more related to DBI architectures involve breaking the transparency (leaking the presence of the DBI, by not taking care of delicate corner-cases) and forcing crashes by playing with DBI leaked internal state.

1.2 Contributions

The contributions of the research include:

1. A series of stand-alone tools and technologies to fast solve specific binary analysis tasks:
    - Creating generic memory dumps, including all memory address space and metadata describing individual memory ranges, which also abstract the target architecture (Windows, Linux, MacOS memory dumps use the same structure, facilitating their offline analysis in a generic way).

[14] https://github.com/buffer/libemu
[15] https://github.com/bitdefender/bddisasm
[16] https://github.com/AmrThabet/x86Emulator
[17] https://github.com/sf2team/vb2014
[18] https://github.com/season-lab/sok-dbi-security

- Correct and fast rebuilding the import table in memory dumps of Windows PE files, taking care of the available modules and disjoint applications running in a target process.
- Fast building and matching of large inter-procedural control flow graphs in binaries, as well as a mechanism to identify the correct process configuration and approximate the Entry Point for each potential application running inside the process, using a brute-force approach.
- A lightweight concolic execution module built on top of a DBI engine written from scratch, adding in the triggering of malicious behaviors in APTs and malicious binaries using analysis evasion techniques, but which could also be used for code coverage and concolic testing.
- A joint-work contribution with Andrei Mogage, Emanuel Onica, Rafel Pires and Pascal Felber, pushing a compiler inside an Intel SGX Enclave to mitigate TEE (Trusted Execution Environments) supply chain attacks for Intel Software Guard Extensions (SGX).

2. CFGDump, a tool based on technologies described in Chapter 2, and able to approximate with high accuracy the deobfusated version for an equivalent process of a polymorphic packed binary (Chapter 3). Despite other process-dump based unpackers which uses Entropy to identify the moment when a process content is unpacked, CFGDump uses a control flow graph fingerprinting mechanism to capture a stable snapshot of the real-time evolving execution flow of the target process. Because CFGDump is designed to target malware binaries, it is aware that inside a process may execute disjoint applications as a result of code injection from other processes. In this matter the process configuration is used to obtain a complete set of disjoint applications, with respect to the set of memory cells on top of which a maximum inter-procedural control flow graph is obtained. For each individual application found, a specific output binary is created, also rebuilding the import table, taking care of the dynamic pointers, which existing tools are not able to handle properly. CFGDump is not only able to operate online on running processes, but also

offline on memory dumps created with the GD tool presented in Chapter 2, making this way the target architecture to be transparent.

3. COBAI (Complex Orchestrator for Binary Analysis and Instrumentation), a DBI engine based environment analysis with a dynamic architecture, ready to automate the analysis of a large number of benign and malign binaries, and providing an increased transparency compared to its predecessors. The dynamic architecture facilitates:
   - analysis of lateral-moving threats and remote execution exploits, through a server-client interface, without requiring human intervention or programming skills; the approach is backed by a configuration file able to adjust parameters for individual plugins;
   - tracking of payloads injected in remote processes, child process creation and event triggered processes;
   - the user choice for a specific disassembly or translation module, if the default ones becomes deprecated, discontinued, new CPU technologies are not supported, or new architectures are targeted;
   - a ready to customize set of plugins (you need a feature you add it, otherwise ignore it); for experienced programmers new plugins can be created, however, by default, existing plugins are able to cover a wide range of use-cases;
   - ability to instrument only the application code, or specific functions (at architectural level), for performance reasons;
   - a high level of transparency compared to other existing DBIs, handling a large number of analysis evasion scenarios, like those implemented in al-khaser[19](> 200 environment artifacts), pafish[20](> 50 environment artifacts), safemachine([10] - 12 DBI transparency issues);

   The engine is presented in Chapter 4.

[19]https://github.com/LordNoteworthy/al-khaser
[20]https://github.com/a0rtega/pafish

1.3 List of Presentations

1. Vlad Craciun, Andrei Nacu and Mihail Andronic, *"It's A File Infector... It's Ransomware... It's Virlock"*, VB2015, "https://www.virusbulletin.com/uploads/pdf/conference/vb2015/Craciun-etal-VB2015.pdf",
   Prague 2015
2. Vlad Craciun and Cristina Vatamanu, *"Win32.Dorbot – A splice between a file infector and a botnet"*, AVAR2015, "http://avar2015.org/vs/agenda/", Danang 2015
3. Vlad Craciun, *"Prospecting ransomware tech"*, BSidesSF, "https://bsidessf2018.sched.com/event/E6jT/prospecting-ransomware-tech", San Francisco 2018

1.4 List of Publications

1. Vlad Craciun, *"Automated analysis of possible malware applications"*, Working Formal Methods Symposium, FROM 2017, publication of category D
2. Vlad Craciun, *"Automated dynamic deobfuscation of static obfuscated binaries (short talk)"*, Working Formal Methods Symposium, FROM2018, publication of category D
3. Craciun Vlad Constantin, Mogage Andrei and Simion Emil, *"Trends in Design of Ransomware Viruses"*, International Conference on Security for Information Technology and Communications, Lecture Notes in Computer Science 11359, Springer 2019, p. 259–272, publication of category C
4. Arusoaie Andrei, Ciobâca Stefan, Craciun Vlad, Gavrilut Dragos and Lucanu Dorel, *"A comparison of open-source static analysis tools for vulnerability detection in c/c++ code"*, 2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), p. 161–168, publication of category C (CORE 2018)
5. Mogage Andrei, Pires Rafael, Crăciun Vlad, Onica Emanuel and Felber

Pascal, *"Supply chain malware targets SGX: Take care of what you sign"*, 38th Symposium on Reliable Distributed Systems (SRDS), p. 52-60, publication of Category A (CORE 2020)

6. Craciun Vlad, Felber Pascal, Mogage Andrei, Onica Emanuel and Pires Rafael, *"Malware in the SGX supply chain: Be careful when signing enclaves!"*, IEEE Transactions on Dependable and Secure Computing, publication of category A^* (according to CNATDCU Standards, https://www.info.uaic.ro/wp-content/uploads/2021/03/evaluare-teze-2018.10.01-acum.pdf, and the UEFISCDI 2020 IF list, https://uefiscdi.gov.ro/resource-821958-if_clasamente2020.pdf)
7. Craciun Vlad, Andrei Mogage, Dorel Lucanu, *"Building deobfuscated applications from polymorphic binaries"*, SPOSE 2021 : 3rd Workshop on Security, Privacy, Organizations, and Systems Engineering (an ESORICS 2021 affiliated workshop, paper submitted on Jul 30, 2021)

2. A toolset for binary analysis

This Chapter presents a standalone set of technologies used by tools described in Chapters 3 and 4, along with a concolic (concrete and symbolic at the same time) execution module to assist binary instrumentation to perform analysis of malicious APTs (Advanced Persistent Threats), and a joint work with Andrei Mogage, Emanuel Onica, Rafael Pires and Pascal Felber to push an Enclave compiler pipeline (compile, link, sign) inside an SGX (Software Guard Extensions) Enclave in order to mitigate some of the software supply chain attacks for Intel SGX build environments.

Building Generic Memory Dumps is a requirement for systems offloading the analysis of running processes to systems capable of obtaining valuable data in a timeless fashion. Existing tools either attempt to rebuild applications from the modules present in the memory of a running process, or create a memory dump ready for debugging a crash dump. If the network where memory dumps are collected, involve a wide range of operating systems, another difficulty should be passed, as memory dumps at the moment are not cross-architecture. Our contribution in this field is to build generic memory dumps, such that memory dumps are cross architecture (this is possible because an application and its corresponding process share the code-memory ranges, no matter the target operating system involved), and also to allow for memory forensics tools to operate on the dump file as would have on the running process. We evaluated the solution, comparing with ProcessDump PD32 and Sysinternals ProcDump tools and found that our solution builds memory

dumps as large as Sysinternals ProcDump, much faster, while PD32 wrongly rebuilt some dumps for the malware data set, making impossible to re-evaluate the dumps.

Rebuilding the import tables in memory dumps is a must when malware process dumps require repairs and fixes, to correct chain the API calls with the code-flow. While there are public available tools to accomplish this, they either damage the import tables in most of the cases (as SCYLLA does in our evaluation), or they are mislead by the wrong Entry Point of unpacked applications, or do not make all the necessary computations to repair all possible imports. In order to correct repair the import tables we first classify the process configuration as a relation between the number of modules and different applications executing inside a target process. The classification identifies the correct configuration as:

- (SM,SA) - Single Module, Single Application: Most benign applications use this configuration;
- (SM, MA) - Single Module, Multiple Applications: Most benign applications having injected payloads from other processes, they have the main module of the benign application but overall totally different applications are running in the same process;
- (MM,SA) - Multiple Modules, Single Application: A configuration used by Emotet and DoppelPaymer malwares, which execute across two different modules (not libraries);
- (MM, MA) - Multiple Module, Multiple Applications: A configuration used by malware creating benign processes and injecting additional modules and payloads, while the benign application executes in its own context, malware also executes in its own context.

The classification of process configuration ensures that we do not relate the target application with API pointers from other applications to build the target application in a wrong way. Another step towards rebuilding the import table consist in identifying the dynamic API pointers (loaded at runtime) as well as merging

import tables in (MM, SA) configurations and while fitting them back to a larger rebuilt import table, instructions referencing these pointers must be patched to relate with the rebuilt import table entries. We compared our solution GCIR (Generic Config-based Import Rebuild) with PD32, PYIATR and SCYLLA and found that SCYLLA damages most of the rebuilt import tables, PYIATR is close to our solution for benign applications and pretty distant for malicious applications (where EntryPoint and import tables are not necessary related to the main module), and PD32 sometimes gets to build an import table but wrongly binds it to the application header (a static analysis tool like IDAPro will see the API pointers as invalid pointers).

The handling of control flow graphs in binary applications in common literature is mostly related to intermediate representations and graph isomorphism. While this approach is quite heavy (both for implementation and for performance), as seen in our evaluation (angr builds control flow graphs at least 200 times slower and with a memory footprint larger than 10GB for large applications), making it unreliable on a large input. Evaluation of ACCDFISA ransomware took more than 1 hour with angr, and only 5.4 seconds with our building algorithm. To have a fast enough building algorithm we reduced the information inside the node structure to its corresponding absolute virtual address and removed the necessity of edge structures by defining the edge as a property of the branch, for the source node. The missing intermediate language also speeds up the entire processing flow as everything takes place on low level instructions. The building algorithm is a slightly shifted standard algorithm to build ICFGs (Inter Procedural Control Flow Graphs), where additional callback nodes and existing code references are added as targets for nodes where the references are present as constant pointers or pointer tables within the source node instructions. To match ICFGs, we designed a fingerprint algorithm able to hash an ICFG structure, to a 64 bit value. The fingerprint uses three abstraction stages:

- an ICFG is reduced to a RTCFG tree like structure, having the same number

of nodes as the ICFG, but with considerable less edges, by removing duplicate edges to already known target nodes;

- the RTCFG structure is traversed in an iterative manner with a DFS algorithm (Depth First Search), building a correspondent traversal array;
- the traversal array is hashed to a 64 bit value.

We use the fingerprint algorithm in Chapter 3 to match successive snapshots (ICFG hash traversals) of a self unpacking process, and consider as the trigger for the dump a fixed number of identical snapshots. We evaluated the fingerprint on two malware families (Evol file infector and Crysis ransomware), using some training and test data-sets and concluded that for applications, where the binary suffers polymorphic/metamorphic node mutations, but the ICFG keeps constant, our fingerprint algorithm is able to fast transform the ICFG, to a 64 bit value, keeping a low rate of false positives as the ICFG size gets larger.

Based on a previous version of COBAI (presented in Chapter 4), we designed a concolic execution module able to execute Windows applications as many times as symbols were possible to introduce on the execution flow. The concolic execution module uses a lightweight semantics on top of the disassembled instructions, and uses Microsoft Z3 as a SMT solver to model side execution paths. A concolic state is used to carry previous solved models to patch future executions, such that any symbol creation will force the usage of a concrete value present in the model, replacing the actual concrete value. To minimize the overhead of the analysis, we only choose to associate symbols with Windows API output values (return values and output parameters). Among all possible APIs, we only consider some as candidates for symbol insertion, as the API returning pointers or allocating memory ranges, require special handling. We evaluated the solution over four demo applications and concluded that on average, while some applications took up to ten executions to exhaust the execution space, the time for a single execution was about one second. In each of the four scenarios, our concolic module was able to capture the maximum execution, revealing the interest path, as three of the four

applications used sandbox evasion mechanisms. The interest path in these cases, was to get to the behavior which successfully passed all sandbox checking.

In an joint work in [5] with Andrei Mogage, Emanuel Onica, Rafael Pires and Felber Pascal, a TCC compiler was reshaped to fit inside an SGX Enclave, in order to mitigate software supply chain attacks, targeting the continuous integration of TEE (Trusted Executed Environments). The contribution to the project consist in refining the compiler input/output such that any dependency would be found inside the Enclave, minimizing this way the attack surface against the compiler pipeline (compile, link, sign).

3. Unpacking polymorphic binaries

CFGDump is a tool built on top of Generic Memory Dumps, able to find an approximate candidate for a polymorphic obfuscated application starting from its corresponding running process, and despite any other process dump and rebuild tool, CFGDump classifies the process configuration using an ICFG bruteforce mechanism, revealing:

- total disjoint ICFGs (uniquely executing applications);
- the Entry Point for each bruteforced ICFG (the address of the root node);
- total memory cells, part of each built ICFGs;
- the moment when the application is unpacked based on monitoring the changes of the ICFG, rather than using a standard Entropy.

Based on the set of largest disjoint ICFGs found, CFGDump builds new applications by reconstructing the import tables according to each individual ICFG, and links the memory cells and the new import tables to a new header. The resulting applications may be statically analyzed, having repaired with a high accuracy, the calls to initial and dynamically loaded API functions, as well as potential constant strings references on the code flow.

CFGDump classifies the polymorphic obfuscation into three layers: entry-level, mid-level and high-level, and pretends to achieve the expected results on the first two of them. Comparing CFGDump with similar tools shows that ignoring the process configuration, leads to wrong dumps like in case of Emotet, where PD32

dumps two different modules with individual import tables, when in fact the two modules are part of the same application, code jumping from one module to another, and the rebuilt import table should have been built considering the memory ranges across both modules, as one.

A great work has been proposed by Preda Dalla in [6] which provides an abstract interpretation orientated towards deobfuscation and detection, and Matthieu Tofighi Shirazi in [18] which elaborates on obfuscation and deobfuscation approaches in Sections 2 and 3. Although our target frame (on malign binaries) lacks the original code or the obfuscator itself, the paper proved to be a good starting point. A few approaches for deobfuscating commercial packers (mostly *entry-level polymorphism*) are presented in [15, 22]. Although they dive into interesting strategies, such as binary instrumentation or various analysis avoidance techniques, our *mid-level* context requires a different or improved approach. As mentioned in *Introduction*, several works[17, 9, 20, 21] tried to attack a similar class of obfuscation than our target. They have been analyzed and, where possible, tested, but did not fulfill our domain of analysis information, making the point of this paper. [19] provides a good example of the effort required on doing a specialized analysis on a type of obfuscator(Themida [2]) and implementation of a solution which targets it. Our focus was to avoid the necessity of creating particular tools and, although a good piece of work, the complexity levels are high when it comes to various types of obfuscators and their versions.

Based on [8] (an analysis environment based on QEMU, capable of recording and replaying executions), Charles Lim et al. present Mal-Xtract in [13] and Mal-Flux in [14], a method to detect the end of unpacking routines based on analyzing a replay of a binary in Panda Qemu environment. The authors evaluate three benign applications on eight packers, among which two are shared by our evaluation (UPX and Themida). Another research to detect the unpacked code using an emulator is provided by Paul Royal et al. in [16], where authors also propose a formal frame for their approach.

4. A DBI for malware analysis

The existing point of view on DBI (Dynamic Binary Instrumentation) engines, makes them frameworks (difficult to be operated by non-programmers) and unreliable to be used on evasive malware samples. The static architectural design in common DBIs brings more difficulties, as updating some CPU instructions inside the dissassembler or operating different some translations, would involve knowing the architecture and the project sub-modules. Given this behavior, an open source DBI, would require skilled people to always maintain the entire project in place of allowing a subset of engine functionalities to be replaced by other components. Another issue related to malware analysis deals with the tracking of the moving code, either locally (from a process to another), or at the network level. There is no known method at the moment allowing reverse engineers in a seamless fashion, to follow the moving code with the existing analysis instance.

COBAI (Complex Orchestrator for Binary Analysis and Instrumentation) is not a DBI framework, but an analysis environment, with a dynamic architecture, a server-client interface to easy track the moving code, using a highly configurable file allowing non-programmers to configure individual plugins, and covering a wide range of transparency from DBI level to operating system level (existing DBIs have issues with some of the DBI transparency levels and as for the OS which is an upper layer, no support at all is provided).

While the tool already comes equipped with a couple of plugins, new plugins may be designed to assist and extend the analysis scenarios. However, the existing plugins are more than enough to produce consistent traces, in a short time. A series

of control flow hooks (to not be confused with detour API hooks) allow COBAI to change the behavior of APIs and low level instructions, either by changing the input, simulating the instruction side effects, or changing the output. The hook plugins also describe a thin layer between the analyzed application and the analysis environment, providing increased transparency not only for the DBI, but for a full sandbox. From our evaluation, COBAI performance is somewhere between DynamoRIO (the best performance), and PIN, and while sometimes gets even lower than PIN, it compensates with lower memory footprints. Comparing the cache performance with PIN, using an old game (blobby volley), COBAI forces the game to operate at 35 FPS (Frames Per Second) using a log file as target, and reaching 57 FPS with suppressed output, while the native game had a constant 62 FPS. PIN on the other hand forced a 40 FPS, equivalent to COBAI 57 FPS (suppressed output).

The transparency of OS level was evaluated on three applications, one being a ransomware, and the other two taken from public available repositories. For al-khaser, COBAI passed 243 out of 244 tests, failing a single WMI (Windows Management Instrumentation) COM Object API (the failed test is related to CPU temperature and in order to pass this test, COBAI must synchronize three different hooks to create and destroy a data-structure which should not normally exist). For pafish, it passed all 54 tests and for Satan ransomware it also passed all 39 tests forcing malware to execute inside a sandbox where it would normally fail to execute. The transparency tests for the DBI were conducted using a series of 12 demos created by AVAST in [10]. COBAI passed 11 out of 12 tests, PIN passed 6, and DynamoRIO passed 4 tests, each of the later two generating a crash on different demos, and failing the oters.

We also conducted a malware analysis test on a series of 10 samples, mostly ransomware binaries, where COBAI finished in at most 8 minutes, applications which would have normally execute forever. Some delay and infinite loop cutting optimizations in COBAI take care of these scenarios, as part of the core analysis, interrupting infinite executions. PIN and DynamoRIO either executed in a larger

time-frame, or crashed (DynamoRIO - Crysis and GandCrab).

5. Conclusion

In this thesis we contribute with some specific binary analysis tools and built on them, other two more complex projects CFGDump and COBAI. CFGDump uses a novel approach to deobfuscate/unpack polymorphic binaries, operating control flow graphs in a different way to match and identify 1 out of 4 possible process configurations. CFGDump also classifies the polymorphic packing into 3 difficulty layers and attempt to provide a solution for the first 2 of them. COBAI is a DBI built for malware analysis, and while its performance lowers sometimes because of the instrumentation and cache performance (compared to PIN and DynamoRIO), it is able to pass DBI and environment evasion techniques, making it an analysis environment ready to face most malware behaviors. The lack of common programatic usage make it ready to operate by software testers and people with less programming skills, compared to framework design in other DBIs like PIN, DynamoRIO, QBDI.

Both COBAI and CFGDump started in 2016-2017 and are both a joint work with Andrei Mogage. Up until now, COBAI shifted 3 times the architectural design, having at the moment a dynamic architecture, which includes core sub-components. While for other DBI engines changing the disassembler or the translator is not possible, COBAI allows for such changes, being suitable for academic research (students can take part in designing optimized translators, instrumentation loops, or analysis plugins).

CFGDump assist binary unpacking and COBAI takes care of new ransomware samples, accelerating the reverse-engineering time, and data extraction (encryption

keys, encryption algorithms, elevation techniques, code injection mechanisms, etc.) at Bitdefender. Both projects take up to 50k LOC and are written using a mix of C/C++, Asembler and Python.

Future Work

We divide our plans for future improvements into separate categories, based on the purpose and type.

COBAI performance: In this direction, our plan is to go much beyond native execution where possible. Therefore, we intend to improve and make more dynamic the caching mechanisms and strengthen the loop analysis (involving taint analysis), in order to better optimize the obfuscated code. Apart from that, we take into consideration the optimization of translation mechanism and callback bindings, such that a maximum performance will be achieved.

Formal requirements: During multiple binary analysis interventions, we concluded that a mechanism to define the requirements may significantly reduce the manual adjustments of parameters. For instance, the parameters for a binary analysis under COBAI, can be unknown, but a corresponding configuration be made concrete, from an abstract definition for the output of the analysis.

Automated tool creation: In order to build a malware removal tool (for persistent, hard to remove malicious applications) or ransomware decryption tool (to decrypt and recover files encrypted by Ransomwares), one must analyze malicious samples and then combine the results with the whole process of software development. We plan to reduce and automate the redundant work by interpreting the results of COBAI's analysis and use them to build templates for such tools (involving taint analysis and concolic execution). Therefore, an analyst could use COBAI to obtain a malicious trace and also be able to generate a removal tool based on this trace, with minimal implication.

Malware classification: Using the executions traces, as well as some other metadata, we may be able to develop classifications mechanisms which will cluster

samples according to various traits and features. This is of special interest for malware detection, where obfuscation is used.

Concolic execution improvements: Having a map of all the application's branches and possible conditions, we may use concolic execution techniques in order to find a proper condition for a different execution path to be activated in malware like Advanced Persistent Threats. In this way, we may force a desired behaviour from an application (e.g., Activate the malicious behavior in sandbox environments, generate a key based on the desired seed, etc). This is different and more generic than our current mechanisms of avoiding detection.

Obfuscation/Deobfuscation: We believe that we can make some advances in obfuscation/deobfuscation field, by involving DBI to strengthen the obfuscation, and also use a static binary translation and taint-analysis mechanism to improve deobfuscation in polymorphic binaries. The CFGDump can also be improved to chart control flow graphs in real-time for interactive analyses, as well as using the CFG fingerprint to trigger detection for the deobfuscated application.

Next level debuggers: As the threats keep evolving, and move from local level to network level, a centralized debugger would be useful to track the moving payloads, not only on other local processes, but also in an isolated network environment. COBAI could provide an API interface for s server-client interactive analysis environment, tracking direct and indirect child processes, payload injection, as well as remote code execution through network administrative shares, or even within exploits like MS17-010[1] used by WannaCry ransomware described in [4].

DBI as a service: given the server-client behavior, COBAI could be used to provide binary instrumentation as a service, facilitating both the access to a centralized server (aggregating individual client analyzes) and the control of lateral movement with dynamic sandboxes, and the split of instrumentation to offload heavy computations on servers with high resources.

[1] https://docs.microsoft.com/en-us/security-updates/securitybulletins/2017/ms17-010

Bibliography

[1] Pin. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.

[2] Themida. https://www.oreans.com/themida.php.

[3] BRUENING, D., AND AMARASINGHE, S. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering . . . , 2004.

[4] CHEN, Q., AND BRIDGES, R. A. Automated behavioral analysis of malware: A case study of wannacry ransomware. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)* (2017), IEEE, pp. 454–460.

[5] CRACIUN, V., FELBER, P. A., MOGAGE, A., ONICA, E., AND PIRES, R. Malware in the sgx supply chain: Be careful when signing enclaves! *IEEE Transactions on Dependable and Secure Computing* (2020).

[6] DALLA PREDA, M. Code obfuscation and malware detection by abstract interpretation. *PhD diss.), http://profs. sci. univr. it/dallapre/MilaDallaPreda_PhD. pdf* (2007).

[7] D'ELIA, D. C., COPPA, E., NICCHI, S., PALMARO, F., AND CAVALLARO, L. Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (2019), pp. 15–27.

[8] DOLAN-GAVITT, B., HODOSH, J., HULIN, P., LEEK, T., AND WHELAN, R. Repeatable reverse engineering with panda. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop* (2015), pp. 1–11.

[9] GUILLOT, Y., AND GAZET, A. Automatic binary deobfuscation. *Journal in computer virology 6*, 3 (2010), 261–276.

[10] HRON, M., AND JERMÁŘ, J. Safemachine: malware needs love, too. https://www.virusbulletin.com/conference/vb2014/

abstracts/safemachine-malware-needs-love-too/, 2014.

[11] JAMALPUR, S., NAVYA, Y. S., RAJA, P., TAGORE, G., AND RAO, G. R. K. Dynamic malware analysis using cuckoo sandbox. In *2018 Second international conference on inventive communication and computational technologies (ICICCT)* (2018), IEEE, pp. 1056–1060.

[12] KIRSCH, J., ZHECHEV, Z., BIERBAUMER, B., AND KITTEL, T. Pwin–pwning intel pin: Why dbi is unsuitable for security applications. In *European Symposium on Research in Computer Security* (2018), Springer, pp. 363–382.

[13] LIM, C., KOTUALUBUN, Y. S., RAMLI, K., ET AL. Mal-xtract: Hidden code extraction using memory analysis. In *Journal of Physics: Conference Series* (2017), vol. 801, IOP Publishing, p. 012058.

[14] LIM, C., RAMLI, K., KOTUALUBUN, Y. S., ET AL. Mal-flux: Rendering hidden code of packed binary executable. *Digital Investigation 28* (2019), 83–95.

[15] MARION, J.-Y., AND REYNAUD, D. Dynamic binary instrumentation for deobfuscation and unpacking. In *Depth Security Conference* (2009).

[16] ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)* (2006), IEEE, pp. 289–300.

[17] SAIDI, H., PORRAS, P., AND YEGNESWARAN, V. Experiences in malware binary deobfuscation. *Virus Bulletin* (2010).

[18] SHIRAZI, M. T. *Analysis of obfuscation transformations on binary code*. PhD thesis, Université Grenoble Alpes, 2019.

[19] SUK, J. H., LEE, J.-Y., JIN, H., KIM, I. S., AND LEE, D. H. Unthemida: Commercial obfuscation technique analysis with a fully obfuscated program. *Software: Practice and Experience 48*, 12 (2018), 2331–2349.

[20] UDUPA, S. K., DEBRAY, S. K., AND MADOU, M. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)* (2005), IEEE, pp. 10–pp.

[21] YADEGARI, B., JOHANNESMEYER, B., WHITELY, B., AND DEBRAY, S. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy* (2015), IEEE, pp. 674–691.

[22] YASON, M. V. The art of unpacking. *Retrieved Feb 12* (2007), 2008.