UCLouvain

École polytechnique de Louvain

Building a mutation tool for binaries

Expanding a dynamic binary rewriting tool to obfuscate malwares

Authors: **Bastien Wiaux, Arnold Gauthier**
Supervisor: **Axel Legay**
Readers: **Ramin Sadre, Charles-Henry Bertrand Van Ouytsel**
Academic year 2022–2023
Master [120] in Computer Science and Engineering

Abstract

New malware are created every day. However, some are just previously known ones, altered to make them invisible to classifiers. But how can a classifier defend itself against them?
Malware mutation tools are developed to help classifiers and to prevent potential malware from accessing a computer. However, these mutators are static, and no dynamic mutation tool exists.
The goal of this work is to create dynamic mutation tools using a dynamic binary lifter and recompiler; *BinRec*. *BinRec* is the only tool capable of dynamically lifting a binary, applying optimizations, and recovering the binary. We will explain in detail how this tool works, and how we use it.
We'll be looking at the 11 different mutation types we have developed, then evaluate each one of them in order to see how well they can change a binary.
Finally, we suggest other mutations to be implemented to further mutate the binaries.

Acknowledgements

We would like to thank our supervisor, Prof. Axel Legay for his help, advice and his encouragement throughout this year.
We also want to express our sincere gratitude towards our assistants, Charles-Henry Bertrand Van Ouytsel, Christophe Crochet and Serena Lucca for their availability, their patience, their feedback, their motivation and their fast answers to every question we had.
Furthermore, we want to thank our readers, Prof. Ramin Sadre and Charles-Henry Bertrand Van Ouytsel for their time to review our work.

On a more personal note, tanks to our respective family, our friends, our roommates, and in particular our partners for supporting us all year long. They've been patient and willing to listen whenever we've talked about any of this.

Contents

Chapter 1

Introduction

In the recent years, the proliferation of malware attacks has raised exponentially, presenting significant challenges to the security of computer systems and networks. With the dynamic nature of malware and the inventivity of their creators to evade traditional security measures, it is essential to find innovative methodologies to counter these evolving threats.

Machine learning classifiers have emerged as powerful tools to counter this threat, by reducing malware detection to classifying programs. However, their effectiveness relies heavily on the availability of extensive and diverse training datasets. Unfortunately, obtaining a sufficiently large and varied collection of labelled malware samples for training purposes presents a considerable challenge. The rapid mutation and customization techniques employed by malware authors continuously exceed the rapidity at which new samples can be collected and labelled.

To get around the shortage of labelled malware samples and to try to win the race, researchers and analysts have investigated a number of strategies, such as manual feature engineering and dataset augmentation techniques. Although these techniques show potential, they are frequently labour- and time-intensive, and they may not be able to detect all malware variants. As a result, there is an increasing need for automated systems that can effectively produce a variety of mutations of current malware samples in order to increase the amount of training datasets accessible.

In this paper, we will introduce a new technique for the automation of malware mutations, using a tool constantly developing and very promising: dynamic binary lifting. It allows us to use any sample of a database, lift it, mutate it as much as we want, and recover a number of variations that will carry the same label as the parent sample. This master thesis is a proof of work, but we strongly believe that if it is continued, it could be a practical solution for the dataset problem of the classifiers.

In order to do so, we will be using *BinRec*, the only tool (to our knowledge) able to lift, apply optimizations, and recover a binary. Once a project has been lifted to an LLVM Intermediate Representation (IR), we will add mutations by modifying the recovered code to hide contents, detect environments, or drown the content in other cleanware. We will then recompile it into another executable that still keeps its original purpose and behaviours.

In order to understand all the concepts that will be used in this work, we will start with a description of the state-of-the-art of the current techniques used in binaries rewriting, malware analysis and mutation. We will talk about the Symbolic Execution for Malware Analysis (SEMA) toolchain, and we will conduct a brief review of the LLVM language that we had to use for our mutations.
After covering the needed knowledge, we will dive into the mutation of binaries, explaining every mutation we developed and automatized.
This will be followed by a quick explanation of how to use the mutation tool, through the CLI and through the SEMA Toolchain WebApp.
Finally, we will evaluate the mutations, first by analysing them one by one, and then by analysing them all together in a full mutation.

Chapter 2

State of the Art

The point of this chapter is to give some background understanding of the concepts used in this master thesis. The first section describes binary analysis, lifting and rewriting using two different techniques. The following two sections are oriented toward the purpose of our work and talk about malware, known mutation tools and how they work. Lastly, we will present the (Symbolic Execution for Malware Analysis) SEMA toolchain that we later used to analyse our results.

2.1 Static and Dynamic Binary Rewriting

The binary rewriting can be used for many applications: post-installation hardening, (de)obfuscation, and reoptimization. But its effectiveness is limited by the complexity and/or absence of the source code [1].
Binary analysis and lifting is an approach to recover information from binaries for which there is no access to the source code. From this information, it is possible to understand the behaviour of the program, and therefore classify it as benign or as a threat. A security analysis can also be done to see if the security policies are held [9]. Binary lifters raise machine instruction to a higher-level intermediate representation (IR), such as LLVM (section 2.5), that we used throughout this thesis.
Binary lifting has not seen wide use because of multiple limitations: it cannot accurately model indirect control-flow targets, differentiate between code pointers and data constants, or identify the boundary between data and instruction bytes [1].

2.1.1 Static Binary Lifting

This section consists of three parts. First, we will talk about the conception of a static binary lifting tool. Then, we will discuss the challenges of binary lifting. And finally, we will see its limits.

Conception of a Static Binary Lifting Tool

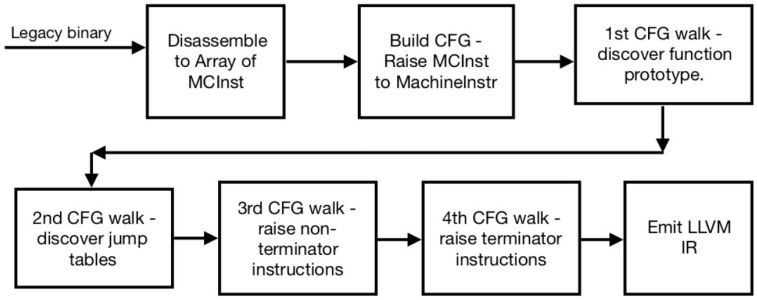


Figure 2.1: Steps in a static LLVM-based raiser [37]

The first step in the conception of a static binary lifting tool is the construction of the Control Flow Graph (CFG) [37]. Once it is generated, a walk of the graph is performed to detect the function prototype. A function prototype "is a declaration of a function that specifies the function's name and type signature (arity, data types of parameters, and return type), but omits the function body" [33].

The second step raises the instructions to LLVM IR. Once we have the function prototype, a second walk of the CFG is performed to construct jump tables, and it is followed by a third walk that raises the instructions of all the basic blocks. Terminating instructions are handled separately. The different types of raised instructions are :

1. Instruction operands: memory referencing operands that point to the stack are raised as stack values, while the rest are either classified as local or global variables by resolving the information in the data section of the binary. Register operands are stored as Static Single Assignment (SSA) values and a map of these values is maintained.

2. Instruction opcodes: each machine instruction is raised as zero, one or more LLVM IR instructions. As said before, during this walk, each of the instructions is raised except for the terminating instructions.

3. Block terminator instructions: blocks with terminator instructions are raised and linked during this final walk.

4. Function calls and global references: while raising instructions, information from the Procedure Linkage Table (PLT) and Global Offsets Table (GOT) is used.

Challenges of Static Binary Lifting

Binary lifting faces two main challenges: recovering the CFG and recovering function boundaries [9].

Control Flow Graph There are two phases in recovering the CFG of a program: identify this basic block composing the application, and establish the correct relationship among them in terms of control flow.
Any binary program analysis must start by locating the portion of the program image containing executable code. When found, the basic blocks need to be identified. The starting addresses of the blocks can be found in multiple sources. The easiest source is to find the entry point of the program, and if we use a dynamic library, the exported functions. Through code analysis, other basic bloc addresses can be identified at different places: jump instructions, constants, and global data are places where pointers to other basic blocks can be stored.

Once the basic blocks, their entry addresses and their size have been collected, the CFG recovery can start. Though direct jump instructions provide useful information, they are insufficient to recover the entire graph. The main issue is handling indirect control-flow transfer instructions, such as indirect function calls or indirect branch instructions. These transfers can be put in three categories [9], allowing the creation of an accurate CFG:

1. Compiler-generated, function local CFG: indirect jump instructions generated by a compiler in order to lower the control flow of certain statements

2. "Reasonable" handwritten assembly: indirect jump instructions manually introduced by the developer in assembly to optimize low-level routines. Reasonable means, for instance, a function that does not make assumptions about the value of its parameters, but enforces the constraints locally.

3. Indirect function calls: indirect function calls via pointers of functions of C++ virtual functions.

Function Boundaries The second problem, recovering function boundaries, consists of identifying the function entry points and correctly associating them with all the basic blocks they can reach, skipping over function call instructions. This problem has multiple challenges:

1. Accuracy of the function boundary: it is highly dependent on the CFG. If the CFG lacks information about the destinations of an indirect jump, the destination basic blocks might not be considered and there would be a loss of accuracy.

2. Deciding if a certain basic block is the entry point of a function or not: a strong indication would be the presence of an explicit function call to its address, but this might not always be there.

3. Call thunks: in computer architectures, if the program counter is not available, a call to the next instruction can be done to retrieve the counter. The destination of these calls should not be interpreted as an entry point.

4. `noreturn` functions : functions that never return, but use `exit` or `longjump`, are sometimes called through a function call instruction. This leads to a wrong path from the call site to the next instruction, which might be part of another function.

5. Shared code: Two functions sharing a portion of their bodies

6. Calls to the middle of a function: a function can sometimes have multiple entry points. This is most commonly seen in handwritten assembly.

7. Tail calls: unconditional jump instructions that have to be handled in a way that prevents them from being identified as a part of the function local CFG.

Limitations

In this section, we describe the limitations of static binary lifting [1].

Code vs Data, Reference Ambiguity The correct label has to be inferred through program analysis to distinguish code from data, and references from constants. However, in general, this problem is undecidable. Therefore, heuristics are used by state-of-the-art analysis tools to estimate the right label set.

Indirect Control Flow Transfer (iCFT) In function of the execution context, iCFTs might transfer control to one or more target locations. In C and C++, function pointers are invoked through the use of indirect calls. Switch statements and position-independent code are implemented by indirect branches. In this code, the offset at which a binary or library is mapped in memory is added to the branch target of all indirect branches that replace all direct branches. Once more, statically identifying all potential targets of iCFTs is undecidable in the general case. When they manage to identify the potential targets of iCFTs instructions that load their destination address from jump tables, static approaches have achieved high accuracy. But a challenge remains: resolving indirect function calls and returns.

External Entry Points Binary analysis can only partially observe the control and data flow between program modules through the interface of external modules when the entire code is not visible. When a code pointer of the main module is passed as an argument to an external module and is used to re-enter the main module, partial visibility can be an issue. Some binary rewriters try to support the issue with special case handlers for the interface of known libraries, but handling external callbacks through unknown interfaces is not correctly done.

Ill-Formed Code Ill-formed code can be caused by aggressive compiler optimizations, as well as manually written assembly code. A common anti-disassembly technique is *overlapping instructions*. Some compilers lower selection control structure as *inline data and jump tables*. The detection of boundaries is complicated by overlapping basic blocks, multi-entry functions, and tail calls.

Obfuscation Binary lifting techniques may face explicitly modified binaries that are intentionally altered to obstruct analysis. In a program protected by an obfuscator (that transforms executable code stored in code sections into bytecode stored in data sections, and embeds a virtual machine into the program to interpret the bytecode), almost no information about the behaviour of the program is revealed. The transformation (and others) can artificially increase the size and complexity of a control-flow graph of a program to a level where static disassembly becomes intractable.

2.1.2 Dynamic Binary Lifting

Throughout this master thesis, we have used *BinRec*, that is as far as we know the only tool for dynamic binary lifting.

Here below are the basic principles of a dynamic binary lifter and lowering tool. Between the two steps, any modification on the LLVM IR would result in a

modification in the final binary.

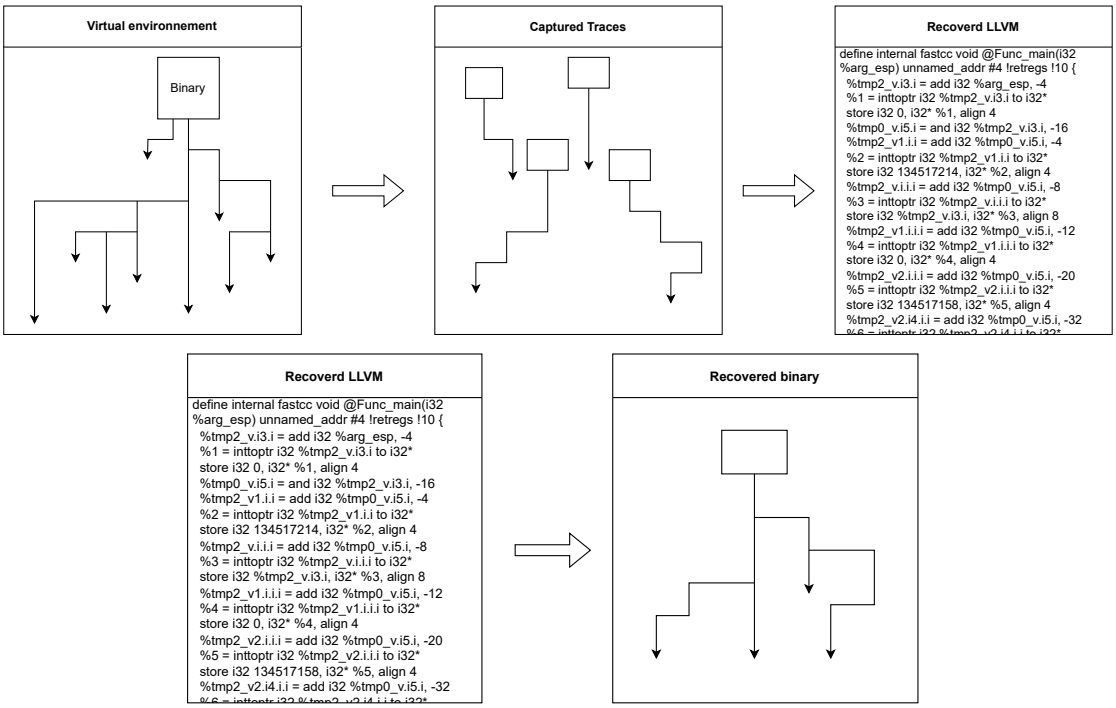


Figure 2.2: Binary Lifting and lowering principle

Conception of a Dynamic Binary Lifting and Recompilation Tool

By design *BinRec* avoids the limitations that we identified for the static binary lifters [1]. This is achieved by leveraging dynamic program analysis to recover accurate disassembly of binaries that are then translated into an IR. Dynamic program analysis involves executing the said program (as opposed to static program analysis, which never runs the file)[32].

Figure 2.3 shows a high-level overview of the approach, which consists of three logical components :

1. a dynamic lifting engine and data collector
2. the canonicalization of the IR code
3. a lowering of the IR code into an executable binary

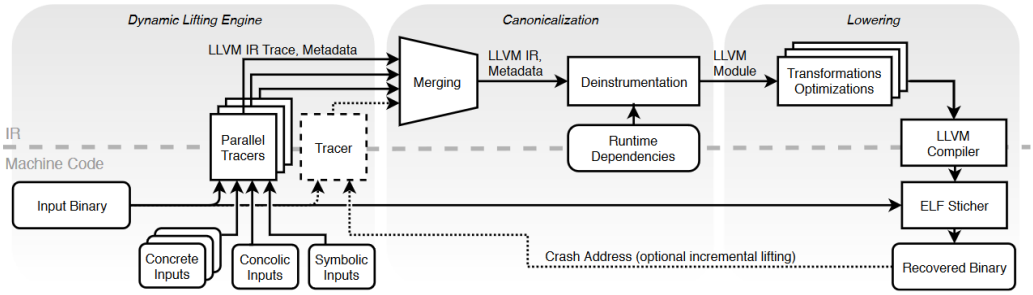


Figure 2.3: The steps of binary recovery: lifting to compiler IR, transformation on the IR, and lowering back to machine code [1].

Dynamic Lifting Engine To drive dynamic execution, *BinRec* takes a multi-pronged approach, using several methods that use various types and sources of inputs. The first source of input should be a test corpus using the desired features, but user-specified tests are unlikely to cover all the code paths to be lifted, as there can be implicit inputs that are less obvious to users but still should be accounted for, such as address layout, timers, etc. A potential solution is to drive execution through the program paths that depend on implicit inputs. In order to do so, some implicit inputs that cannot be triggered through explicit inputs are simulated. For instance, if the control flow depends on the virtual address layout, the implicit source is checked by altering the length of environment strings so that the space layout is altered.
If the user needs to explore every path of the program, *BinRec* supports concolic execution, and can also take fuzzer-generated, concrete inputs to drive the binary lifting frontend. The input generation interface is flexible and extensible, allowing the dynamic driver to be customized.

Dynamic Data Collector Dynamic data about the execution of each program path is recorded. This data is essential to overcome the limitations of static binary lifting that we explained earlier. *BinRec* records which instructions were executed, where the function boundaries are, and the observed targets of each branch instruction. This is used to accurately disassemble the binaries and produce canonical IR. The recorded data are completely accurate on covered paths, but it is not possible to reason on data that are not covered by the dynamic traces.

Canonicalization By using different execution driving paradigms, *BinRec* can compose program traces generated over different runs. It uses a technique to merge all distinct traces into one specialized program that will behave correctly on all

covered paths. One unique LLVM IR module is created from several using the metadata that was collected during the lifting phase. The code of the combined program is the union of all basic blocks seen in the merged traces. This procedure is path-insensitive. The outputted CFG looks like the original one but lacks the nodes that were not visited during the lifting.
By using an emulation-based dynamic lifting engine, *BinRec* is able to lift programs compiled for a different instruction set architecture compared to the host system. But the IR generated from this engine is heavily instrumented to make the execution in a virtualized environment easier. The *deinstrumentation* component of *BinRec* eliminates dependencies and merges all captured code together in one LLVM IR file that can be used and compiled into a standalone binary.

Lowering Once the IR has been modified as required by the client, a functional recovered binary is produced. *BinRec* uses an unmodified LLVM compiler (llc) to generate a temporary ELF file from the recovered binary. After, the lowering toolchain puts together sections of the original binary and the recovered binary into one other binary. Most of the sections come from the recovered binary. At last, binary patching is executed to insert the trampolines to support external callbacks and update dynamic linking structures. "Trampolines [...] are memory locations holding addresses pointing to interrupt service routines, I/O routines, etc. Execution jumps into the trampoline and then immediately jumps out, or bounces, hence the term *trampoline*." [36]

Challenges of Dynamic Binary Lifting

BinRec faces two major challenges: the coverage of the code, and the scalability [1].

Coverage The first challenge is to drive through all desirable code paths. The optimization and the security hardening will not require the same paths: the former just needs the most frequented executed paths, and the latter might just need paths accessed through trusted inputs and will neglect the rest. The path covered by *BinRec* depends on the inputs. *BinRec* can also merge traces into a single LLVM IR module that therefore recovers multiple paths. However, every path might not always be covered, which might lead to the execution of code that wasn't lifted, which is called a *flow miss*. These misses can happen when the flow of the program depends on information such as a random number, timing, or literal memory addresses. *BinRec* deals with these via customizable miss handlers based on application scenarios.

Scalability The second challenge is scalability. Generating inputs to achieve maximum coverage is difficult and can lead to a path explosion for complex

programs. *BinRec* can record multiple independent traces of binaries, which can be merged at a later stage.

Limitations

BinRec has 3 major limitations. First, it only handles single-threaded x86 ELF binaries, because of the engineering effort required. Second, it does not handle self-modifying code because it would take some more lifting time, and it would complicate the traces merge. Last, it does not recover a mapping between stack slots and variables, even though it would improve optimization and allow more fine-grained instrumentation.

2.1.3 Dynamic vs Static

We compare here dynamic and static binary lifting and rewriting, following the results of Altinay et al. [1].

In addition to solving all the limitations mentioned in 2.1.1, the quantitative advantages were also analyzed. *BinRec* was compared to its static equivalents, McSema [10] and Rev.ng [9]. The comparison was limited to active, open-source binary lifters which aim to be compiler-agnostic.
Only a few binaries were recovered by McSema. During the attempts to run binaries without optimizations, issues were faced with McSema's handling of double-precision floating-point operations in 32-bit applications, unsupported xmm instructions on 64-bit, and segmentation faults in the C++ delete operator. Also, segmentation faults were caused by some binaries lifted from compiler-optimized code upon launch, or produced an incorrect output. As IDA Pro is used by McSema for control graph recovery and analysis, McSema is limited due to IDA's inability to accurately recognize function pointers in real-world code.
Concerning Rev.ng, most of the dynamically linked binaries were not recovered. For instance, the output of the tool differed from the output of the original file when trying to lift *libquantum*. The only tests that were correctly recovered were statically linked, and *BinRec*'s performance of the output was better than the of rev.ng as we can see in Figure 2.4.
In summary, both tools tested against *BinRec* were unable to recover even standard binaries, even though they are widely used state-of-the-art tools.

	BinRec		McSema		Rev.ng
Benchmark	**O0**	**O3**	**O0**	**O3**	**reported**
400.perlbench	1.25	1.48	–	–	3.7
401.bzip2	0.76	1.05	2.84	–	2.2
403.gcc	1.26	1.37	–	–	2.1
429.mcf	0.83	1.00	2.31	1.41	1.5
445.gobmk	1.04	1.56	–	–	3.3
456.hmmer	0.77	0.74	–	–	2.2
458.sjeng	0.77	1.08	3.43	–	2.6
462.libquantum	0.95	1.30	2.07	1.04	1.1
464.h264ref	0.80	1.24	–	–	2.7
471.omnetpp	1.92	3.09	–	–	2.8
473.astar	0.80	0.94	–	–	1.5
483.xalancbmk	1.12	1.66	–	–	2.8
geomean	0.98	1.29	–	–	2.25

Figure 2.4: Measured execution time normalized to the original binaries. [1].

2.2 Malware Analysis

2.2.1 Malware Definition

Any script or binary that has a malicious purpose can be referred to as a "malware". They come as executables, shell code, script, or firmware. Here are some definitions for "malware":

- "[...] as the name suggests malware are intended to harm computers and computer users by stealing information, corrupting files, or by just doing mischievous activities to annoy users."

  — ("A study on malware and malware detection techniques")[28]

- "Malware is a collective noun denoting programs that have a malicious intent [...]. Specifically, malware usually denotes hostile, intrusive, or simply annoying software programmed to gather sensitive information, gain access to private systems, or disrupt legitimate computer operations in any other way."

  — ("Towards automated malware creation: code generation and code integration")[6]

- "A program that is inserted into a system, usually covertly, with the intent of compromising the confidentiality, integrity, or availability of the victim's data,

applications, or operating system or of otherwise annoying or disrupting the victim."

— (*Information Security*)[27]

Those definitions are very similar in what they say. They all include the notion of harm or malice and the notion of acting without the user's consent or knowledge.

Categories of Malware

They are various types of malware. Some embed themselves in other software, some spread over the networks or encrypt all the data of the computer in the hope to obtain a ransom. [34]

Virus A virus is a piece of software that hides itself inside a cleanware. It usually copies itself and spreads in other programs or files. Viruses are likely to perform harmful actions such as destroying data and sometimes causing hardware high stresses.

Worm A worm is a malware spreading itself over the network to infect as many computers as possible. It is a standalone file that does not need user interaction. Once the spread over a network has begun, it is autonomous.

Rootkits Rootkits is a tool used to modify the host's operating system to hide the malware. It can be very powerful, hiding harmful processes from the user.

Backdoor A backdoor is designed to give an attacker access to resources that they should not have. It is a piece of software that is usually used in combination with another type of malware to bypass the protections and then activate itself to make the access to the victim's computer sustainable.

Trojan The purpose of the Trojan is to entice its victims to install it or perform an action that will actually launch a malicious operation. To do this he mimics benign programs and usually spread via social engineering. They are different types of Trojan, we can here cite the most interesting.

> ***Remote Access Trojan (RAT)*** Many modern forms work in conjunction with a backdoor. They contact a central server and can then gain unauthorized access to the infected computer. They can install additional software such as a keylogger to steal confidential information, cryptomining software or adware to generate revenue for the Trojan operator.

Droppers Their only goal is to deliver malware onto the host they infected. The key idea is that the Trojan is easier to deliver without detection, and when in place it can open the way to other bad things.

Ransomware They are the most publicized malware, ransomware hold the host computer hostage. To do so, they can use two ways: locking down the computer or encrypting the data. The last kind is called crypto-ransomware.

Our tool could in theory take care of all kinds of malware with perhaps greater difficulty for the viruses. However, the practical limitations of the lifter prevented us to confirm this statement.

This is only theoretical due to the practical limitations of the lifter, which will be discussed later on.

2.2.2 Analysis

Just like binary lifting, there are two different types of malware analysis: static analysis and dynamic analysis.

Static Analysis

Static analysis is analysing malware without executing it [29]. Static analysis can be done by disassembling the binary or by scanning for specific strings [5]. Before doing any analysis, the executable has to be unpacked and/or decrypted. The debugger and memory dumper tools can be used to reverse compile executables. This analysis is the only practice sufficiently fast to be used for on-access detection, which is why this is used by antivirus software. However, binary obfuscation techniques that modify the binaries are designed to resist reverse compilation. These techniques change the syntactic properties of the binaries and make it harder to analyse while keeping the same behaviour.

Dynamic Analysis

Dynamic analysis is analysing malware while executing it in a controlled environment, such as a virtual machine, a simulator, an emulator, a sandbox, etc [29], to prevent it from spreading. This technique is used to address obfuscation strategies that affect the malware's syntactic features but do not change its behaviour [5]. There exist various techniques to perform dynamic analysis, like function call monitoring, function parameter analysis, information flow tracking, etc. Dynamic analysis does not require decompiling or disassembling the malware and is more effective than static analysis, but it is time and resource consuming. However,

malware can have sandbox detection techniques to check whether they are being executed in a sandbox, and will exhibit a different (non-malicious) behaviour and often delete itself. Moreover, the malware can be triggered by certain events, such as an exact date or a specific command, and wouldn't be detected in an artificial environment. Norman Sandbox, CWSandbox, Anubis, TTAnalyzer, Ether and ThreatExpert are some online automated tools for dynamic analysis of malware. These tools generate analysis reports that provide an in-depth understanding of the malware's behaviour and offer valuable insights into their actions.
There are three components in a dynamic analysis framework [22]:

- **Malware sample**: An executable code, a script, a document, or firmware.
- **Hardware and operating system** : Specific OS to execute the malware. If the malware isn't in the required conditions, it will not execute.
- **Analysis tool**: To monitor the analysed code/system, a software or a device is needed. The tool should produce a report to summarize the malware's behaviour. Even though higher abstraction is better to classify the sample (determine if it's malware or not, and the category of the malware), lower abstraction is needed to completely understand the behaviour and the techniques used by the malware.

2.3 Existing Mutation Tools for Malware

Previous studies have already been conducted on the subject, laying the foundations for different kinds of mutations. Here is an overview of the different techniques.

2.3.1 API Call Addition

Application programming interface (API) calls are often used to classify the behaviour of a program, and are a common choice to train malware classifiers [25]. Attacks such as API Call Addition can compromise the integrity and availability of machine learning classifiers and algorithms.

Rosenberg et al. [25] demonstrated an attack on binary classifiers that are used to distinguish benign and malicious API call sequences. They modified a correctly classified malicious sequence so that it was considered by the classifier as benign after transformation. Without lifting the binaries, they modified the sequence and added API calls using a mimicry attack. Mimicry attacks "can be defined as attacks that achieve attacker-intended effects without modifying aspects of an

application behavior that are monitored by an IDS" [23]

Fadadu et al. [12] prepared a list of 300 new API calls instead of generating configuration files per malware, like Rosenberg et al. The new APIs have the same name, input and output as the original APIs. The goal is to replace the original to make an extra call during the execution of the malware. They call the original API first and save the return value, and then run benign code while keeping the needed data safe. The value is then returned and the next step of the original malware is executed.

Kucuk et al. [20] assume access to the source code of the malware. They use obfuscation to hide API calls to prevent API feature extraction tools from finding a selected list of APIs that are used in malware.

2.3.2 Code Injection

Kruegel et al. [19] developed a technique to evade detection features of detection systems by allowing an attacker to execute system calls in their correct execution context. By manipulating code pointers, an attacker can obtain and release control of the application's execution flow.

2.3.3 Other Techniques

Process Chopping Ma et al. [21] presented AutoShadow, a tool that evades behaviour-based malware detectors by automatically splitting a piece of malware into multiple "shadow pieces". These processes do not contain recognizable malicious behaviour, but together, they can still fulfil their original purpose. These approaches could easily be used with signature-evading techniques to evade signature-based and behaviour-based detectors.

De Gaspari et al. [8] developed an attack that splits malware operations into a set of cooperating processes so that none of the split processes is flagged as malware. The attack that they proposed can be used in the ransomware domain. The ransomware is run and split more and more until it is not detected by the classifier anymore.

Program Chopping Ispoglou et al. [17] proposed malWASH, a dynamic diversification engine that, without being detected by analysis tools, executes an arbitrary program. The behaviour of the original program is hidden behind many processes that exhibit benign behaviour. This is achieved by dividing the target programs into smaller chunks that are executed within the context of other processes. They

are later connected together through a scheduler that also transfers states between the different processes. However, malWASH requires admin privileges to work correctly.

Pavithran, Patnaik, and Rebeiro [24] created D-TIME, which has the same purpose as malWASH. But unlike malWASH, D-TIME is threadless and doesn't require administrative privileges to execute. They instead use pre-existing benign threads during random outbursts. It makes D-TIME stealthier and gives a wider usability since it can be used with normal user privileges.

Genetic Programming Castro et al. [7] presented (Automatic Intelligent Malware Modifications to Evade Detection) AIMED, a program that automatically finds optimized modifications using a genetic programming (GP) approach. When implemented in a previously detected malware, it results in a misclassification. However, adversaries using this approach will face a tradeoff between finding an evasion manually or increasing the time spent waiting for one automatically. AIMED works only on static malware.

2.4 SEMA-SCDG

The tool we will use to analyse malware is the SEMA toolchain [2]. SEMA stands for Symbolic Execution open-source toolchain for Malware Analysis [3], and the architecture of the toolchain is shown in Figure 2.5. In this section, we will focus on SEMA-SCDG, a tool used to build a System Call Dependency Graph (SCDG).

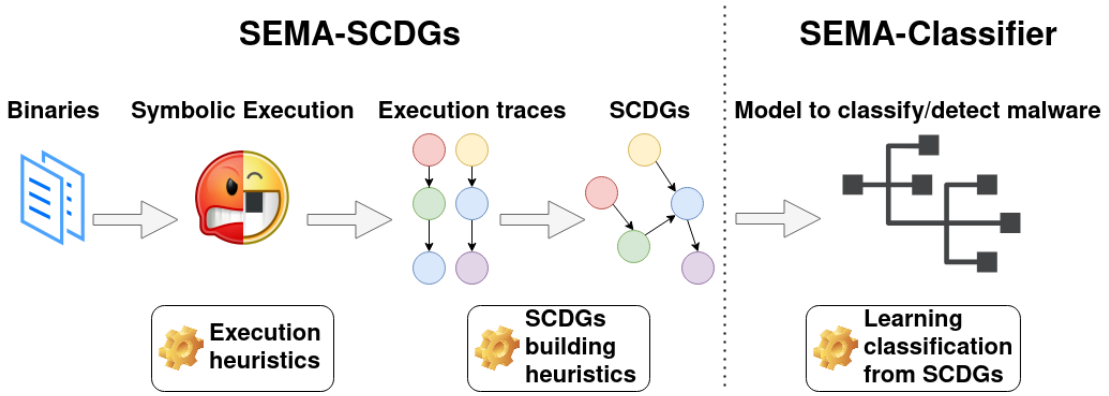


Figure 2.5: Architecture of the SEMA-Toolchain [2].

2.4.1 Binaries

The first step in creating a SCDG consists of collecting different binaries from different families. The toolchain can handle ELF and PE files, except for .NET files or malware that display control flow manipulation or virtualization.

2.4.2 Symbolic Execution

The binaries are symbolically executed through *Angr*. All possible exploration paths of the binaries are explored without a dynamic execution of the code nor assigning specific values to the variables [4].

An *accuracy/performance tradeoff* is faced by the symbolic execution [31]: a lot of resources are needed (time, memory, ...) but the more resources are available, the better the performance will be. However as there is a problem of state explosion, the resource usage must be optimized and *Angr* needs help to be as efficient as possible when exploring state space.

To optimize the resource usage, at each step, a limited subset of paths is looked for execution, while the other ones are placed in a stash and will be used late when more space is available.

The correct modelisation of the OS and external libraries called during execution is another big issue of symbolic execution. A mismodeling of these libraries can result in the creation of incorrect states or failure to explore significant states.

2.4.3 Building SCDGs

Once the binary has been explored through symbolic execution, we obtain different execution traces. Each one consists of a list of API calls and their arguments, their resolved address, and their calling addresses. To build the SCDGs, all of this information is used. As in Figure 2.6, in the graphs, the system calls are represented by vertices, information flows between the calls by edges, and the orders in the traces are represented by the directions of the edges. It is possible to identify the family of malware by observing similarities in their SCDGs.

2.4.4 SCDG limitations

The SCDG's are limited from multiple points of view.

Limitations of SCDG itself SCDGs are prone to attacks exploiting the way they are built and exploiting how they define edges and vertices.

If two arguments of calls have the same value, the SCDG will create a link between them. This could be used to introduce fake links between unrelated parts,

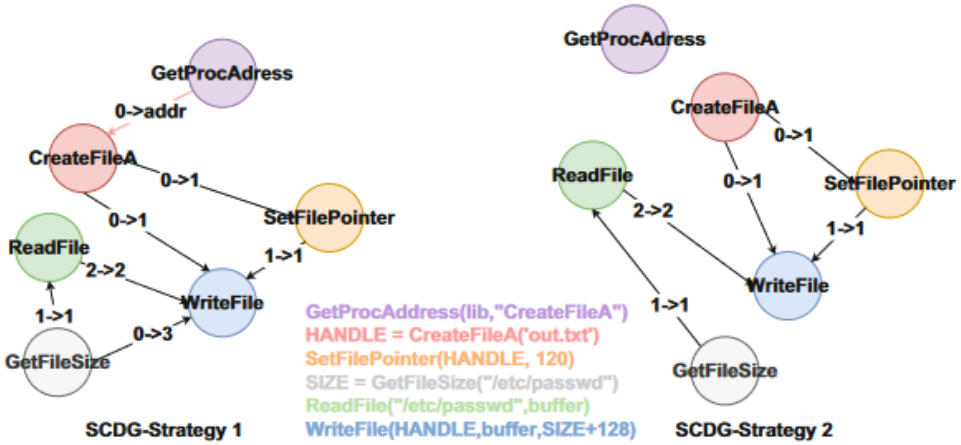


Figure 2.6: SCDGs built with two different strategies [4]

or even "transplant" introduce a completely new SCDG to the original one, using the variables as stitches to link them.

The vertices of the SCDG represent system calls, so if an attacker replaces a system call with an equivalent one, the signature would be completely different as the vertex would have changed.

Symbolic Execution Limitations Since the SCDG is built from the observations of the symbolic execution, it inherits all the limitations of this execution. This means that the SCDG could be inaccurate because of [35]:

- **Path explosion** The scalability of symbolically executing all feasible program paths diminishes when dealing with large programs. As the program size increases, the number of feasible paths grows exponentially, and in the case of programs with unbounded loop iterations, it can even be infinite.
- **Memory aliasing** This difficulty arises when multiple names can access the same memory location, known as aliasing. In such cases, it is not always feasible to statically recognize aliasing, which leads to the inability of the symbolic execution engine to acknowledge that modifying one variable also impacts another.
- **Arrays** When dealing with arrays, symbolic executors face a choice between considering the entire array as a single value or treating each array element as an individual location. However, the challenge arises when treating array

elements separately, as references like "A[i]" can only be dynamically specified when the value of "i" is known concretely.

2.5 The LLVM language

During our experiments and research, we manipulated a lot of LLVM code. While being widely used by language creators, only a few people are directly coding in LLVM, the standard being using C++ or C as an interface. For this reason, very few resources are dedicated to the grammar and semantics of this language. We mainly collected our knowledge from the *Language Reference* [16] and from the *Command Guide* [15] for the compilation of the code.

Here we summarize the key principles we have learned in the hope of helping newcomers to get started more quickly and not reproduce our struggles.

2.5.1 Variable Declaration

There are two types of variables: local and global ones. The main difference is that global variables are allocated at compile time, while local ones are allocated at runtime. Note that we are talking here about allocation and not initialization, meaning that a global variable could be declared and only initialized with values during runtime. A variable can be marked as constant with the keyword `constant`, which is a very important feature for global variables, as it enables the compiler to store them in read-only sections and optimize the memory layout. Of course, if a variable is marked as constant, it cannot be initialized nor changed afterward. A variable cannot be assigned more than once.

The Declarations

Global Variables The global variables take an "@" as prefix of their name and can be marked by a variety of modifiers such as `internal`, `unnamed_addr` or `common`. The first is similar to `static` in C, the variable will show as a local symbol in the final ELF. The second means that the address of the variable does not matter, what matters is the content. The last one may not have an explicit section, must have a zero initializer, and may not be marked as `constant`. For a complete overview, please refer to the documentation in "linkage type" and "global variable".
Here is an example of a global variable declaration.

```
1 @str = constant [13 x i8] c"I am evil!!!\00", align 1
```

Local Variables The local variables take a "%" as a prefix before their name. There are two types of local variables (rigorously referred to as *local identifiers*).

> The named variables are much like what we are used to in other languages, you can name them as you want, as long as it is a unique identifier.
>
> Unnamed variables are trickier to work with because they are identified by a number and this number must always verify a predicate: the unnamed variables must follow the numerical order, with no skipping allowed. This order is defined following the file lines, not the code flow. This does not seem like a strong predicate, but if you are modifying a code, it is crucial to know it.

The local variables are assigned the result of an operator, meaning that `%i = i32 0` is invalid, you must instead do `%i = add i32 0, 0` or `%i = xor i32 0, 0`.

2.5.2 Functions

Functions in LLVM are very similar to those in other languages. They are declared using the `define` keyword.

```
1 define internal fastcc void @Func_main(i32 %arg_esp) unnamed_addr #4
  ↪ !retregs !10 { ... }
```

The function body is kept inside of braces. When the function must return a value, the keyword `ret` is used.

As you can see, functions are global, and like global variables, they need a "@" as a prefix. To call a function we use the `call` instruction followed by the function name and its arguments between parentheses.

```
1 tail call fastcc void @Func_main(i32 ptrtoint (i32* getelementptr
  ↪ inbounds ([4194304 x i32], [4194304 x i32]* @stack, i32 0, i32
  ↪ 4194300) to i32)) #7
```

External functions can be called in LLVM. To do so, you need to declare it without a body and then declare the external function in the import table or link the bytecode containing the function to the bytecode of your LLVM. The latest will be explained in Section 3.1.5.

2.5.3 Parameter Attributes

The parameter attributes are used to communicate additional information about the result or the parameters of a function [16]. These attributes are considered to be a part of the function and not a part of the function type.
Parameter attributes are keywords that follow the type specified. If more than one parameter is needed, they are separated with a space.

When *BinRec* lifts binaries, like for instance a basic `Hello_world.c` program (as in Appendix A.1.1), it writes the function attributes like so :

```
1 ...
2 attributes #4 = { norecurse }
3 attributes #5 = { nounwind readnone }
4 attributes #6 = { nobuiltin nounwind "no-builtins" }
5 attributes #7 = { nounwind }
```

And when the attributes are needed in a function, *BinRec* can just call the variable to use its content.

```
1 define internal fastcc void @Func_main(i32 %arg_esp) unnamed_addr #4
  ↪  !retregs !10 {
2 ...
3 }
```

Which is exactly the same as replacing `#4` by its value:

```
1 define internal fastcc void @Func_main(i32 %arg_esp) unnamed_addr
  ↪  norecurse !retregs !10 {
2 ...
3 }
```

2.5.4 Metadata

In LLVM, the metadata is used to give extra information about the program to the code generator and the optimizers. The metadata can be added to instructions and global objects. Metadata are defined using an exclamation point "!". Still using our previous example, we can see that *BinRec* lifts the metadata like so:

```
!0 = !{!"clang version 14.0.0"}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
...
```

Code 2.5.1: Metadata Declaration

```
define internal fastcc void @Func_main(i32 %arg_esp) unnamed_addr #4
↪  !retregs !10 {
...
}
```

Code 2.5.2: Metadata Call

Unlike the Parameter Attributes, the metadata variables cannot be replaced by their value.

2.5.5 Conditions and Jumps

The conditions and the jumps can be compared to the conditions and jumps in assembly language. It has two steps:

1. Compare two elements and store them in a variable (Line 1).
2. Depending on the previous result, jump to the first or the second block (Line 2).

The blocks that the execution will jump to must have names, and are defined like in Line 3.

```
%.not8.i.i = icmp eq i32 3 , %rand_fin7
  br i1 %.not8.i.i, label %next8, label %BB_4
next8:
```

Chapter 3

Mutation of binaries

In this chapter, we develop every mutation we have implemented in binrec. We have split them in 5 categories: mutations on strings, the addition of cleanwares, the workflow hacking, the addition of API calls and escaping debugging and test environments.
The whole project is available at `https://github.com/wiauxb/binrec-tob`, in the mutator folder.

3.1 String mutation

The goal of this mutation is simple. Obfuscate the strings of a program such that any detection techniques based on a string signature would fail. The strings are most of the time stored in the `.RODATA`, and sometimes are in the `.TEXT` section.

In the `.RODATA` the strings are stored "as is" which means all the bits are in the right order, next to each other, in clear, and ready to be analysed.

The `.TEXT` case is a little bit less obvious but can be easily predicted. The string is in the `.TEXT` because the code is pushing a hardcoded string on the stack. This means that we can observe the argument to the instruction **`push`**. The string is therefore cut into parts of 32 bits and the bits are ordered following the endianness of the binary, often little endian.

3.1.1 Overall strategies

From the observations above, we decided to try two strategies to apply our string mutations:

1. Remove the string from the `.RODATA` and hard code it directly in the code, in the `.TEXT`.

2. Make the mutation “in place” in `.RODATA` and adapt the addresses used by the code in the `.TEXT` accordingly.

3.1.2 Details before diving in

Before explaining the mutation process in itself, we need to address some important topics.

String Mutation selection

The principles we use could mutate the strings in any fashion, as long as there is a C function that can deobfuscate dynamically the string. But for the purpose of our experiments, we had to keep a clear framework and choose a limited number of mutations that clearly demonstrated the potential and versatility of this process. This is why we choose to implement the following obfuscations:

- Splitting / Shuffling
- XORing
- Base64 encoding

The splitting clearly demonstrates manipulation of data capabilities, XORing is a primary operation for all computers. And finally, the base64 encoding shows more complex "encryption" capabilities. The process behind base64 encoding is a bit more complex and could theoretically be extended to any encryption scheme.

String Detection

To be able to mutate the strings we first need to find the string. If the string is directly hardcoded in the LLVM code, it is a trivial task. But if the code loads it from memory, it is more complex to find the right addresses that point to a real string. To achieve this, a first approach is to follow all the addresses referred to in the code and to determine if they were in the `.RODATA` or the `.TEXT`. This is a good first filter, but it appears not to be narrow enough.

> If we declare integer constants in the code, they would be stored in the `.RODATA` and would be labelled as string while they are not.
>
> If there are jumps in the code, the addresses to which they would jump would be stored in the `.TEXT`, being also identified as a string.

Example Here is an example of storing a string address into the variable `%6`

```
store i32 134520840, i32* %6, align 16
```

The variable will later be loaded

```
%arg.i.i.i = load i32, i32* %6, align 16
```

and this way the value stored at `134520840` can be used through `%arg.i.i.i`. The problem arises when differentiating this line

```
%. = select i1 %.not.i.i, i32 134520867, i32 134520878
```

from this one

```
%.6 = select i1 %.not.i.i, i32 134517306, i32 134517326
```

While the first one is selecting between two strings, the second one is selecting an address to jump to, it is part of a conditional jump. The two string addresses point to the `.RODATA` while the other ones point to `.TEXT`.

You now understand why good string detection is crucial to differentiate very different cases that use very similar LLVM code.

Solution Correctly identifying strings is very important as the manipulations we would do on the strings suppose that the string is `null` terminated, and integers or addresses are not, which would cause our modification to erase and move too much from the original binary.

Our current take to minimize the errors and keep the consistency of the binary no matter what is the following: we continue to check the section to which the address is pointing. On top of that, we read the data from the address to the first `null` byte. If all the bytes in this range are valid Unicode characters, we consider the address to point to a string. Otherwise the address is discarded.

This is, of course, not a perfect solution for finding strings, but it is an easy and good enough solution for finding "mutation-compatible" bit strings, which is in the end what matters the most to keep the executability and consistency of the binary.

3.1.3 `.RODATA` section manipulations

In this manipulation, we want to modify a string stored in `.RODATA` and adapt the references in the code to the new "way of storing" the string. Doing so by manipulating the LLVM recovered file is quite straightforward.

Get the string First thing first, we need to get the string we want to mutate and then to remove it from the original binary. This way when *BinRec* links the recovered binary to the original one, the string will not be reintroduced.

Mutate the string and store it Once the string is retrieved comes the interesting part. We will modify the string in Python, applying whatever mutation we have chosen. Bear in mind that we only apply one mutation at a time, going through the whole process before doing another mutation. Once we have our mutated string, it is ready to be stored in the new `.RODATA`. To do so we can use the global variables of LLVM such as the ones that are generated when lifting a program:

```
@fpstt = internal unnamed_addr global i32 0
@fp_status.0 = internal unnamed_addr global i8 0
@stack = internal global [4194304 x i32] zeroinitializer, align 16
@onUnfallback = common local_unnamed_addr global i1 false
```

```
; Replace: store i32 134520840, i32* %6, align 16
@str.1.1 = constant [7 x i8] c"vil!!!\00"
@str.0.1 = constant [6 x i8] c"I am e"
```

Code 3.1.1: A splitted string example

So if our mutation is splitting the string in two, we will write Code 3.1.1 or if we want to XOR the string, Code 3.1.2. We can see in Code 3.1.2 that the variable `@str.1` has "^^Y" and "^^[" characters. Those are non-ascii chars that don't show well in editors but they are fine and intact in the file. They are the result of the XORing of `@key.1` on the original string.

When declaring our global variables, we are defining the order in which they will be written in the future binary. This is very important since it gives us the ability to reorder the parts of the string when we are splitting it. It means that

```
; Replace: store i32 134520840, i32* %6, align 16
@str.1 = private unnamed_addr constant [13 x i8]
 ↪  c"x^^Y^^[^PS/1^^XCNs\00", align 1
@key.1 = private unnamed_addr constant [13 x i8] c"19z3p6YXtboR\00",
 ↪  align 1
```

Code 3.1.2: A XORed string example

if we split the string into characters, we can scramble it as desired. This is very powerful! It means that we can shuffle all the strings of a binary with a new random permutation each time.

Dynamically retrieve the original string To keep the integrity of the binary, we now need to implement in LLVM a way for our program to retrieve the original string. To tackle this task we first code an LLVM snippet ourselves such that it works for a basic example. We can observe in Code 3.1.3 an example of this. After ensuring the snippet is properly working, we code a Python function that will generate the LLVM text in a modular fashion i.e. adapting to the number of splits, loading from more variables, adapting the lengths of the tables, etc. An example of the result of such code can be seen in Code A.2.1.

```
@.str2 = constant [7 x i8] c"vil!!!\00"
@.str1 = constant [6 x i8] c"I am e"
```

```
%sp2.1 = alloca [13 x i8]

%s0.1 = load [6 x i8], [6 x i8]* @.str1
%sp2.1.1 = bitcast [13 x i8]* %sp2.1 to [6 x i8]*
store [6 x i8] %s0.1, [6 x i8]* %sp2.1.1
%next0.1 = getelementptr [13 x i8], [13 x i8]* %sp2.1, i32 0, i32 6

%s1.1 = load [7 x i8], [7 x i8]* @.str2
%sp2.2.1 = bitcast i8* %next0.1 to [7 x i8]*
store [7 x i8] %s1.1, [7 x i8]* %sp2.2.1

%spi1 = ptrtoint [13 x i8]* %sp2.1 to i32
store i32 %spi1, i32* %6, align 16
```

Code 3.1.3: Handwritten code to load handwritten split

To insert this snippet, we need to be precise and link this code correctly to the old one. Our goal is to replace the exact instruction line storing the address of the original string into a variable by our line that will store the pointer to the char array obtained by our code.

```
; This
store i32 134520840, i32* %6, align 16
; becomes this
store i32 %spi1, i32* %6, align 16
```

Now that we

1. Found the string
2. Mutated it

3. Stored it

4. Added code to dynamically retrieve it

We can recompile and link the LLVM code using *BinRec* and "voilà !" the binary is mutated and the string does not appear in clear anywhere.

3.1.4 `.TEXT` section manipulations

The manipulation towards storing the mutated string in the `.TEXT` is very similar to the manipulation described in the section 3.1.3. The key difference resides in the storing part. Instead of storing the string in a LLVM global variable, we will directly declare the variable in the code of the main function, as a local variable.

This means that the data of the string will end up in the `.TEXT`. However, this way of doing things is less straightforward than the `.RODATA` one.

Current Limitation Actually, since the string is now a part of an instruction, its format and shape are bounded to the requirements of this instruction. This means that we cannot shuffle the different parts of the string any more, the order of the instruction is imposed by the way we need to push on the stack.

The optimization at compilation time is also limiting this technique. Indeed, currently, the XOR and base64 obfuscations are optimized during the recompilation/linking phase such that in the final binary, the key remains present but unused and the XORed string is optimized to its clear form. This makes these mutations completely ineffective.

These limitations make the `.TEXT` manipulation not ready yet to be called effective, but we strongly believe that if means are found to circumvent them, this will be a very effective automatic obfuscation technique.

3.1.5 The base64 challenge

To achieve a base64 mutation, the first 3 steps stay the same. We can find the string, mutate it with Python and store it back in `.RODATA` or `.TEXT` the way we described here above. But the challenge arises when we tackle the last part. Decoding base64 is an easy task conceptually but when it comes to implementing it in LLVM, it is exactly as you would expect from writing in such a low-level language, very long, verbose and utterly hard to debug. We got down to the task and developed a decoding function which can be found in Appendix A.5. However, it never worked and we dropped it with the following thought.

This function could eventually have worked, but it would have been a lot of effort for a little achievement. If we had managed to pull it off, we would only get

the capacity to decode base64 ciphers and nothing more. But what we were looking for was a generic and easy way of using any encryption ... And so we came up with a simple but extremely powerful idea: calling arbitrary C functions directly from LLVM. If we were able to do it with system calls, it was possible for sure to extend the technique to arbitrary code.

And that's how we do it! To decode the base64 cipher generated in Python, we call a custom function called `base64_decode` and written in C.

```
; declaration of the external function
declare i8* @base64_decode(i8*)
; using the function in the code
%plain.ptr.1 = tail call i8* @base64_decode(i8* %cipher.ptr.1)
```

In the LLVM part, this is all that we need. For the C function, the steps are the following.

First, we need to compile the code. Be sure there is no main function in your C file as it would interfere with the main of the lifted code. We want the LLVM bytecode of the C function, so we will use Clang to do so.

```
clang-14 -m32 -emit-llvm -c <C files> -o external_functions.bc
```

It's worth noting that we use clang-14 as all the *BinRec* framework works with LLVM v14, and we compile with the -m32 flag to produce 32-bit compatible instructions.
Now that we have our bytecode it is time to link it to the bytecode of our mutant. LLVM has a tool for that, and using it is very straightforward.

```
llvm-link-14 recovered_to_link.bc  external_functions.bc -o
↪  recovered.bc
```

It will finally give us a complete `recovered.bc` that we can feed back into *BinRec* to recompile and link with the original binary.

To ease the process even more, we created a simple command to link and recompile a *BinRec* project after mutations:

```
just link_recompile <project> <files to link, separated by spaces>
```

this will produce a `custom_recovered` just as the `just recompile` would give.

3.2 Clean code Addition

Clean code addition allows a user to add a chosen number of cleanwares to a project they want to mutate. The addition of cleanwares is done in three steps. First, we found some basic C code [14] that we compiled with the flags `-no-pie -m32` to follow *BinRec*'s requirements. We put the binaries through *BinRec*, recovered them, and kept the resulting lifted `recovered.ll` files. Second, the `recovered.ll` files are modified to only keep information that can be useful, such as the variable definitions and the function definitions. Third, when we want to add a cleanware, we get the corresponding file and alter it into the project. Once it is done, the cleanware can be added to the project. We will develop the last two steps.

3.2.1 Cleanware From Recovered Files

We need to modify the `recovered.ll` files that we recovered previously, in order to only keep the useful information and discard the rest. The pretreatment goes in 3 phases.

1. Find every attribute and their value
2. Replace the attributes variables with their values
3. Generate the cleanware

Finding and Replacing the Attributes

To find the attributes, we created a regex rule capable of finding whether a line was the declaration of an attribute. Once an attribute was found, its name and value are kept in a tuple in order to be reused. Once the end of the file is reached, the declarations of the attributes are removed from the cleanware.
The purpose of this action is to avoid duplication of variable names. The declaration helps make the code cleaner, but is not necessary. Replacing them seemed to be the best and easiest solution. Once an attribute is seen in a line, it is replaced with its value.

Generating the cleanware

Once the attributes have been removed, we have to generate a copy of the recovered project and save it. To correctly insert a cleanware, we replace the name of the `Func_main` function with the name of the file.
The last thing added to the cleanware is the correct way to use the call. It will be later used to implement the cleanware into a project. Every cleanware is stored in the `mutator/adder/cleanware` folder.

3.2.2 Adding Cleanwares Into a Project

Once the cleanware have been created, we can add them to a project we want to mutate. The project must have already been recovered with *BinRec*.
This part also works in 3 phases:

1. Change the metadata in the cleanware
2. Find the needed references in the cleanware
3. Add the found references in the project we want to mutate.

Changing the metadata

Like the attributes, the metadata are always declared using the same name from one project to the other, as shown below. However, unlike attributes, the metadata cannot be replaced in the code by their value. The solution was to change the name of the metadata. At first, we wanted to replace them with random strings, but the metadata had to be unnamed. So we split the problem in two. First, we find the number of metadata in the project that we want to mutate. Second, we add the said number to every declaration or call of metadata in the cleanware. Here is an example:
Let's say we want to mutate a basic program that prints "Hello World!" After recovering the program, we see that the file has 11 metadata, going from 0 to 10 (`!0 ... !10`). We want to add a random cleanware, here `print_2d_arraym15161635.ll`, that has 14 metadata (`0! ... !13`).

```
!0 = !{!"clang version 14.0.0"}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{i32 1, !"NumRegisterParameters", i32 0}
!6 = !{!7, !7, i64 0}
!7 = !{!"double", !8, i64 0}
!8 = !{!"omnipotent char", !9, i64 0}
!9 = !{!"Simple C++ TBAA"}
!10 = !{!11, !11, i64 0}
!11 = !{!"int", !8, i64 0}
```

```
!12 = !{i32 0, i32 0, i32 0, i32 0, i32 0}
!13 = !{!"printf"}
```

After the process, the metadata from the cleanware will go from `!11` to `!24`.

```
!11 = !{!"clang version 14.0.0"}
!12 = !{i32 1, !"wchar_size", i32 4}
!13 = !{i32 7, !"PIC Level", i32 2}
!14 = !{i32 7, !"uwtable", i32 1}
!15 = !{i32 7, !"frame-pointer", i32 2}
!16 = !{i32 1, !"NumRegisterParameters", i32 0}
!17 = !{!18, !18, i64 0}
!18 = !{!"double", !19, i64 0}
!19 = !{!"omnipotent char", !20, i64 0}
!20 = !{!"Simple C++ TBAA"}
!21 = !{!22, !22, i64 0}
!22 = !{!"int", !19, i64 0}
!23 = !{i32 0, i32 0, i32 0, i32 0, i32 0}
!24 = !{!"printf"}
```

Finding the References

The next step is finding every reference in the cleanware that we might need to insert into the project. We noticed 4 different types of references :

1. Functions declaration
2. Variables declaration
3. Metadata declaration
4. Function call we inserted earlier

Add the References

Once all the references have been found and stored, we look at whether each reference has already been defined or not in the project. If not, we add them after the `@Func_main` function of the recovered project. As a last step, we add the cleanware call into the `@Func_main`. The code is then ready to be recompiled.

3.3 Workflow Hacking

To hack the workflow of malware, we acted in two steps. First, we implemented "basic if" additions that are always true, and print any message that the user inputs. Second, we modified what we had previously done so that the condition was random, and would exit if the random condition is not met.

3.3.1 Basic if condition

The basic if condition we implemented works as follows:
First, we generate the condition and the block the code is going to jump to. To make the condition always true, we check that 0 is equal to 0. As it is true, the code will jump and print a message that can be determined by the user. If the user enters more than one message, there will be one if condition for every message.
Second, we add :

1. The declaration of the strings we want to print (Code 3.3.1)
2. The declaration of the `puts` function if it has not been defined yet (Code 3.3.2)
3. The if condition at a random place in the code's main function (but always before the last return of the function) (Code 3.3.3)
4. The block that the condition will jump to (Code 3.3.4)

Here is what the different parts look like when implemented in a basic "Hello World" program that we lifted. The whole program is available at Appendix A.3.1

```
1 @.str1 = private unnamed_addr constant [8 x i8] c"Premier\00"
2 @.str3 = private unnamed_addr constant [9 x i8] c"Deuxieme\00"
3 @.str5 = private unnamed_addr constant [11 x i8] c"Troisième\00"
```

Code 3.3.1: Example of variables to print

```
1 declare dso_local i32 @puts(i8* noundef) local_unnamed_addr #3
```

Code 3.3.2: Declaration of the `puts` function

```
  %.not2.i.i = icmp eq i32 0 , 0 ;always true
  br i1 %.not2.i.i, label %BB_2, label %next2
next2:
```

Code 3.3.3: Condition and jump

```
BB_2:
  %cast2= getelementptr [8 x i8], [8 x i8]* @.str1, i64 0, i64 0
  call i32 @puts(i8* %cast2)
  br label %next2
```

Code 3.3.4: Block and jump back to code

3.3.2 Random if condition

For the random if condition, we compare two random numbers (one of which is decided at runtime). If the numbers are different, we exit the program. If they are the same, the program continues as normal, until it reaches the next condition. We implemented the mutation on a basic "Hello World" program that we lifted, and where the user chooses to add 2 random conditions, with a max value of 3(A.3.2). The mutation is added to the project as follows:

1. We add the declaration of the functions that we need if they are not already in the lifted project (Code 3.3.5)
2. We generate a number that will be random at runtime, with a max value declared by the user (Code 3.3.6)
3. We create a random if condition that will check if the previously generated number is the same as another random number, but this one will always be the same (Code 3.3.7)
4. We make the block jump if the condition is false. If it is false, we exit the program. (Code 3.3.8)

The user can choose the number of if's they want to add to a project, as well as the max value of the random comparisons. The probability to reach the end of a program is :

$$(\frac{1}{max\_random\_value})^{number\_of\_if's}$$

```
declare i32 @rand() local_unnamed_addr  noinline
declare void @srand(i32) local_unnamed_addr  noinline
declare i32 @time(i32*) local_unnamed_addr noinline
```

Code 3.3.5: Function needed to generate random numbers

```
%time1 = tail call i32 @time(i32* null)
tail call void @srand(i32 %time1)
%rand_init6 = tail call i32 @rand()
%rand_fin7 = srem i32 %rand_init6, 5
%rand_init2 = tail call i32 @rand()
%rand_fin3 = srem i32 %rand_init2, 5
```

Code 3.3.6: Generation of two random numbers modulo 5

```
  %.not4.i.i = icmp eq i32 4 , %rand_fin3
  br i1 %.not4.i.i, label %next4, label %BB_4
next4:
```

Code 3.3.7: One of the two conditions generated

```
BB_4:
  ret void
```

Code 3.3.8: The return bloc

3.3.3 System Call Replacement: puts and printf

In this section we want to show that it is possible to change a system call with an equivalent one to generate a different SCDG, but without altering the output of the program. We chose to replace the `puts` function with the `printf` function. The whole example used here under is available at Appendix A.4.1.
It works in three steps:

1. We generate new metadata and the function call if it is needed for the print instruction (Codes 3.3.9 and 3.3.11)
2. We generate the new lines that will replace the function call in the main. An extra print must be added for the line return that is initially in the `@puts` function (Code 3.3.10)
3. We add and replace everything in the project.

```
@.str1 = private unnamed_addr constant [1 x i8] c"\0a"

declare dso_local i32 @printf(i8* noundef) local_unnamed_addr  naked
 ↪  noinline "frame-pointer"="none" "no-builtins"
 ↪  "no-trapping-math"="true" "stack-protector-buffer-size"="8"
 ↪  "target-cpu"="pentium4"
 ↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
 ↪  "tune-cpu"="generic"
```

Code 3.3.9: Variable and function declaration

```
1 %fp2 = tail call i32 @printf(i8* nonnull dereferenceable(1) %10)
  ↪  nobuiltin nounwind "no-builtins" , !funcname !11
2 %cast2= getelementptr [1 x i8], [1 x i8]* @.str1, i64 0, i64 0
3 %11 = tail call i32 @printf(i8* nonnull dereferenceable(1) %cast2)
  ↪  nobuiltin nounwind "no-builtins" , !funcname !11
```

Code 3.3.10: Calls to replace the puts function

```
1 !11 = !{!"printf"}
```

Code 3.3.11: Extra meta-data needed

3.4 API Call Addition

The API call addition can be compared to the cleanware adder in its functioning. We first need to generate the calls that we want to implement, and then we have to insert them into the project that we want to mutate. The difference with the cleanware adder is that in this case, we use previously obtained results as arguments for the API calls.

3.4.1 Generate the Calls

To generate the calls, we looked into previously created C programs [14]. Once the programs are done and compiled (always using `-no-pie -m32`), we run the recover command on them using *BinRec* to figure out how to use the system calls in LLVM. We manually saved the lines we needed (function declaration and function call, argument and return types) into `mutator/adder/sys_call_llvm.txt`, as shown in 3.4.1

```
1 declare i32 @htonl(i32) local_unnamed_addr #4
2 = tail call i32 @htonl
3 i32,int
```

Code 3.4.1: Example of API call generation

The first line represents the declaration of the system call, the second is how to call it (we will later talk about the variable names and the argument insertion), and the third represent the return type, and the argument types.

3.4.2 Insert the Calls

Find the Variables

Before being able to insert the API call into the project that we want to mutate, we need to find variables we could use. To do so, we go through the project and find strings and potential integers. To find the strings, we use the same function as the string obfuscation. To find the integers, we take every integer that is not the address for a string. Therefore, these are only "potential" integers. We have no way to be sure that these are actually used, or would be references to something else.

Insert the API Calls

The API call insertion is straightforward:
First, we select a random call we want to insert, that has not been inserted yet.
Then, we insert the definition of the API before the `@Func_Main` function.
To follow, we insert the call into the `@Func_Main` function.
If the API needs a variable, we will randomly choose one that was already declared.
If we need to use a variable that stores a string, the string is first loaded in another variable before being used.
Finally, we randomly choose a line between the beginning and the end of the main function, and we add the call to this line. The line must be after the declaration of the variables used in the added call.

3.5 Escaping Debugging and Test Environments

In this section, we explore the three principles that we set to demonstrate the capabilities of our technique to react to the environment of the program. For

demonstration purposes, the only reaction we implemented is an exit from the program, but of course, this can be extended to any desired behaviour.

3.5.1 Environment variables

Our first trigger is the presence of an environment variable. In this example, we will use the DEBUG variable, but of course, in real-life situations, you would search for specific variables to see if a specific software is present, an environment is set up, etc.

This detection is quite easy thanks to the `@getenv()` syscall. We just need to store the string of the variable name we want to search for and call `getenv` on its pointer.

The getenv syscall returns a pointer to the value of the variable in the environment, or NULL if there is no match. This is what we will use, if there is no match, we let the execution follow, or else we jump to the end of the main function, skipping all the malicious behaviours.

This piece of code is injected at the beginning of the main function.

```
; Detect debug variable
  %debug.str.0.1 = alloca [6 x i8]
  store [6 x i8] c"DEBUG\00", [6 x i8]* %debug.str.0.1
  %debug.ptr.0.1 = ptrtoint [6 x i8]* %debug.str.0.1 to i32
  %result.0.1 = tail call i32 @getenv(i32 %debug.ptr.0.1)
  %must.escape.0.1 = icmp ne i32 %result.0.1, 0
  br i1 %must.escape.0.1, label %.escape.1, label %.proceed.0.1
.proceed.0.1:
...
.escape.1:
  ret void
```

3.5.2 Tracing detection

Detecting if we are traced is very similar to the environment variable detection that we just discussed. The only twist is that we're gonna use a different syscall: `@ptrace()`.

The `ptrace` function takes four arguments but the only one that interests us is the request argument, the first one. It indicates how we want to trace the tracee, in this case, we will use the `PTRACE_TRACEME`, which value is 0. The other arguments

are irrelevant when using `PTRACE_TRACEME` so we set them all to 0. When we call `ptrace` on ourselves, the process will only succeed if we are not already traced. In that case, it will return -1 instead of 0. This way we can observe if we are already the trace of someone and in that case, we stop our execution.

```
; Detect tracing running
  %result.1 = tail call i32 @ptrace(i32 0, i32 0, i32 0, i32 0)
  %must.escape.1 = icmp eq i32 %result.1, -1
  br i1 %must.escape.1, label %.escape.1, label %.proceed.1
.proceed.1:
...
.escape.1:
```

3.5.3 Detection of VM

As we encountered for the base64 case in the string obfuscation part, the detection of a virtual environment is proof that our technique is capable of introducing more complex behaviours into a binary.

For this task, we will reuse the external call trick (CFR. section 3.1.5). For this demonstration, we used a basic code found on GitHub [18] (Appendix A.6) and adapted it to our needs. The code performs three tests and if at least two come back positive, we can assume we run under a vm. The code is only suited to VMWare, but any other VM detection code could be adapted the same way.

The `@detect_vm()` function returns an int acting as a boolean, 1 is a positive guess that we are in a vm, and 0 is the opposite.

```
;----------------------------
; Detect if we are running in a VMWare vm
  %result.1 = tail call i32 @detect_vm()
  %must.escape.1 = icmp ne i32 %result.1, 0
  br i1 %must.escape.1, label %.escape.1, label %.proceed.1
.proceed.1:
...
.escape.1:
;----------------------------
```

Chapter 4

Using The Mutator

All the different mutations we've seen in the Chapter 3 are the building blocks of our tool. To use those blocks, we created multiple ways to get one mutation, a handful of mutations on one file, or even generate a plethora of mutants from one original binary. All of this can be done through two mediums:

- The CLI
- The SEMA-ToolChain WebApp

The CLI gives full access to the possibilities that we implemented, from little precise mutations to mass generation, all the power is in your hands. Regarding the toolchain, we didn't have the time we'd hoped for to implement a full interface. However, the key features are there, as we will explain here.

4.1 Through the CLI

BinRec uses `just` to connect with the terminal. To let a user use our mutator, we decided to add our content to the previously existing `justfile`.
The command goes as follows:

```
1 mutate project *flags:
```

Afterwards, we collect the flags and give them as input to a Python script, `auto_mutation.py`. It is important to note that this `just` command is only responsible to execute one mutation on the LLVM representation of the binary. After running it, you still need to recompile the LLVM to get the binary (CFR. 4.1.2). If you want to chain the mutations, the command `just get-mutants` is what you are looking for (CFR. 4.1.3).

4.1.1 The arguments of `mutate`

The flags are different for every mutation:
`strings` has itself three sub-commands:

1. `split` : takes a number of cuts to perform `-ncuts`
2. `XOR`
3. `base64`

All the commands can take the `-text` flag, which does the mutations on the txt section of the project as well

`sys_adder` takes the number of system calls `-number_add` that the user wants to add to the project.

`clean_adder` also takes the number of cleanware `-number_add` that the user wants to add to the project.

`basic_if` takes as argument a list of words that the conditions will print.

`random_if` takes two arguments :

1. `-max_random`: the maximum value for the random if conditions.
2. `-number`: number of if conditions the user wants to add to the project.

`replace_puts` doesn't need any flags.

`escape`, just like strings, has three sub-commands as well:

1. `envvar` is the only one that has an extra flag, `-var_name`, which is a list of environment variables to escape
2. `traced`
3. `vm`

4.1.2 The `recompile` commands

In order to compile our mutants, we created two new commands.

recompile project *flags This command first takes the recovered.ll file that was mutated in the project, and compiles it into a bc file with the `llvm-as-14` command.
After that, we use our custom python script `compile_recovered.py` to compile the previously obtained file into an executable called `custom_recovered`.

link_recompile project *files This second command is exactly like the first one, but it has an extra step before recompiling into an executable. Thanks to the `llvm-link-14` command, we are able to link extra files to our file. This command allows us to add functions in the llvm files that are defined externally.

4.1.3 The `get-mutants` command

This command is a synthesis of all the previous commands. Its purpose is to generate as many (random) mutants as the user wants. It takes two arguments:

1. The name of the project to mutate
2. The number of mutants the user wants to generate

The mutants are stored in a mutations folder inside the Selective Symbolic Execution Platform (s2e) project. When executing this command, three types of files are created:

1. A .txt file that contains the commands that were executed for every mutation
2. the mutants in the llvm format
3. the recompiled mutants

4.2 Through the ToolChain WebApp

To use the SEMA WebApp, you need to run the following:

```
$ make build-web-sema
$ make run-web
```

This will run the Python server from which we will operate.

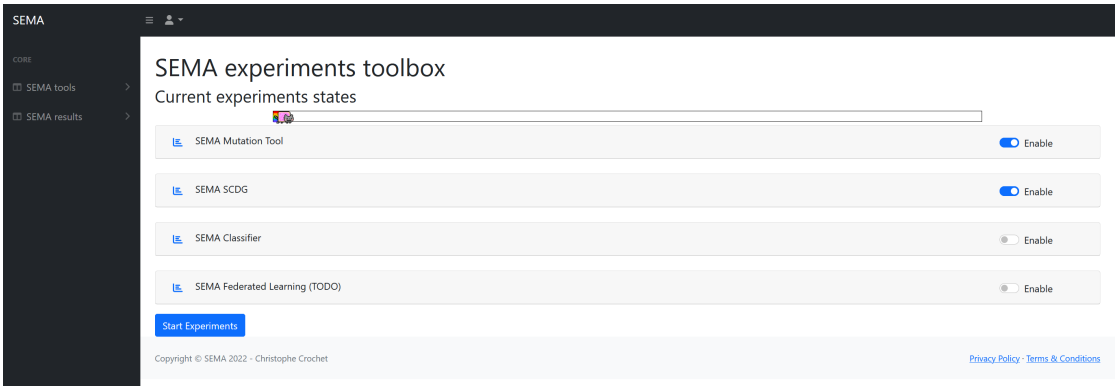


Figure 4.1: The Web-SEMA interface

4.2.1 Structure of the toolchain

The toolchain, and especially the web app, are yet under development, so we had to work our way through the code to establish a place for our mutator.

The toolchain web app is composed of different modules which will each handle a specific task. From top to bottom, we can see the mutation tool (our tool), the SCDG builder, the classifier, and finally a future module for federated learning. The design is pleasing and seems well-partitioned, but under the hood, the code was messier.

The Arguments

The problem with the arguments comes from the way the web interface is built. The whole page is a single HTML form, causing all the different arguments to shuffle into one huge json request. This causes multiple problems:

1. If two modules use the same argument name, this will cause a clash between the two, resulting either in an Exception or into a merge of the two-argument values into one array.

2. The code in the server needs to re-analyze all the arguments to determine for whom they are intended.

3. As they are present in the form, no matter if the module is enabled or not, the arguments of those disabled modules were misparsed.

To tackle those problems without re-implementing all the frontend, we did multiple changes. First, for the "parsing when disabled" and "same name arguments" problems, we re-implemented the `ArgumentParserTC` to enable it to activate or

deactivate the different parts linked to the modules. To see the files CFR to Appendix B.

The process is the following: when the server gets a POST request, meaning that the user pressed "*Start Experiment*", it will first check which modules are enabled. Only for those modules, it will extract the right arguments from the whole request and enable the corresponding modules in the parser. Now that it has isolated the interesting arguments and selected which parser to analyze them, it will run the parser and then run the experiment as usual.

It is important to understand that with this new parser, each module gets only the arguments meant to reach him, which reduces greatly the risk of bugs.

4.2.2 Mutation Module

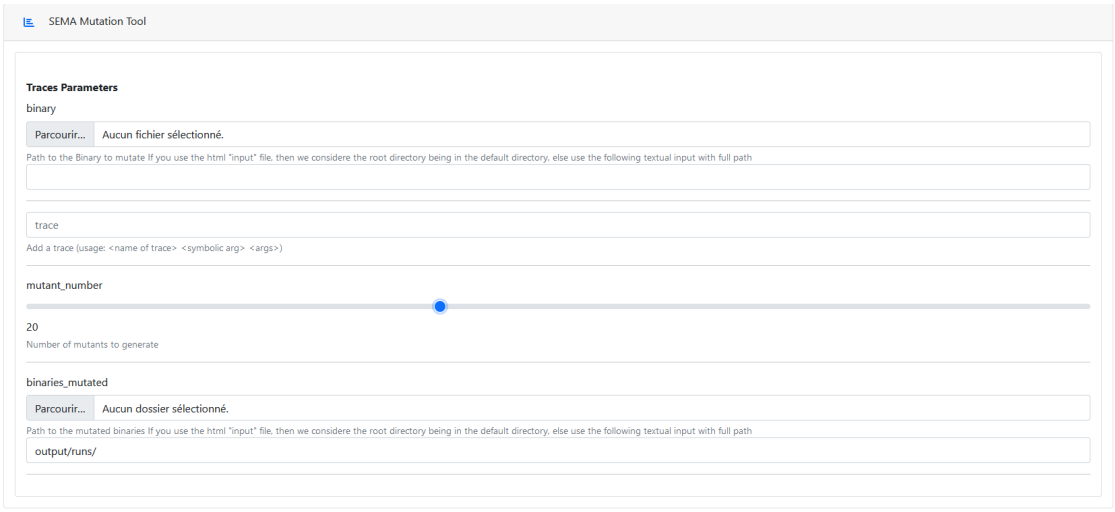


Figure 4.2: Mutator Web Ui

The mutator module takes 4 different inputs. They all are mandatory.

- *binary* is the path to the initial binary
- *trace* gather the trace arguments for the binary lifting
- *mutant_number* is the number of mutant binary to generate
- *binaries_mutated* is the folder to save the mutations to

From those inputs, the module will generate the requested number of mutants, exactly as the `just get-mutants <project> <number>` would do.

4.2.3 Interactions

Here, we will describe the interaction between the Sema toolchain and our tool.

The backend of the WebUI does not directly run our tool. Instead, it communicates with a docker container through a REST API. This API is quite basic but has all the necessary entries to execute the web app tasks.

/add-project With a POST request, you can create a *BinRec* project with the name `project` and the binary path `path`. The `project` and `path` variables are given as the json body of the request.

/projects/<project>/add-trace Also with a POST request, this path lets you create a trace for `project`. The arguments are transmitted through the json body, there are four: `trace_name`,`trace_args`,`trace_symbolic_args`, and `trace_stdin`.

/projects/<project>/recover Recovers the `project`. The recovering part is made of 4 steps.

- It runs all the traces saved previously. Each run is done symbolically by s2e if necessary.
- It merges the captured traces from the precedent steps. This uses a combination of static and dynamic analysis data to try to make the best choices.
- It lifts the merged trace to LLVM IR, using all of what we've seen before. It also directly re-compiles the LLVM recovered for the following step.
- It validates the recovered binary by comparing the different outputs of the recovered versus the original binary, using the known traces.

/projects/<project>/mutate It is the exact equivalent of the `just mutate` command, taking `project` from the URL and the other arguments in the json body as one string in the variable `args`.

/projects/<project>/mutations/<num_mutants> Last but not least, this one runs the mutant generation. It only takes the `project` and `num_mutants` from the URL. The output dir is not yet supported, all the mutants are saved in the **`s2e/projects/<project>/mutations`** directory.

Chapter 5

Evaluation of the Mutations Generator

To evaluate our mutations, we created a basic project that has 3 functions, each one being executed depending on the input. The code is available at Appendix C.1.

1. Prints "I am evil!!!"
2. Opens a file and writes "I altered the file!" in it
3. Creates a socket, and closes it. We were not able to add more socket functions because of the limitations of *BinRec*.

The program was compiled with the flags `-m32 -no-pie` to be able to be put through *BinRec*.

5.1 The Tools

To evaluate the efficiency of our mutations, we used two important tools: Ghidra and the ToolChain. We also used Yara for specific signature forensics. These tools help us analyze the control flow and system calls of the mutant binary and get an overall image.

First, Ghidra is a helpful and well-known tool for reverse engineering. It allows us to see if the instructions and code structure are still close to the original binary. With Ghidra, we can see how the program controls its flow and if different parts of the code were still identifiable.[11]

Next, we rely on the ToolChain to create the SCDG. As this represents the system calls made by the binary and how they relate to each other, we can observe how the binary interacts with the operating system and how the different calls relate to each other.

Last, we employ Yara, a tool that helps us search for specific patterns or signatures within the binary. This tool is useful for identifying known malicious behaviors or vulnerabilities. In our case, by defining rules with Yara, we can look for strings or behaviors we know from the original binary and see if they last into the mutants.[13]

Before analyzing the mutations, let's take a look at what the graphs of the original binary look like.
On one hand, on the SCDG 5.1, we can see the links between the system calls. The important elements are different system calls used. The `strcmp` functions and the `fopen`, `fwritefclose` functions are linked together because they use the same variable. The seven paths in the main represent every if comparison, and the path where all conditions are false. We obtained the following results from the toolchain:

- <SimulationManager with 4 deadended>
- Total number of blocks: 62
- Total number of instr: 314
- Number of blocks visited: 53
- Number of instr visited: 240
- Syscalls Found:'malloc': 8, '__libc_start_main': 6, 'strlen': 10, 'strncmp': 3, 'strcmp': 3, 'printf': 1, 'fopen': 1, 'socket': 1, 'fwrite': 1, 'close': 2, 'fclose': 1
- Number of nodes: 12
- Number of links: 14

The four deadends represent every possible way to reach the end of the program, and we can see all the system calls that were used in the project.

On the other hand, the graph 5.2 from Ghidra shows us the possible paths possible. We can clearly see the four different paths that depend on the input (three of them lead to other blocks, and one leads to the end of the program).

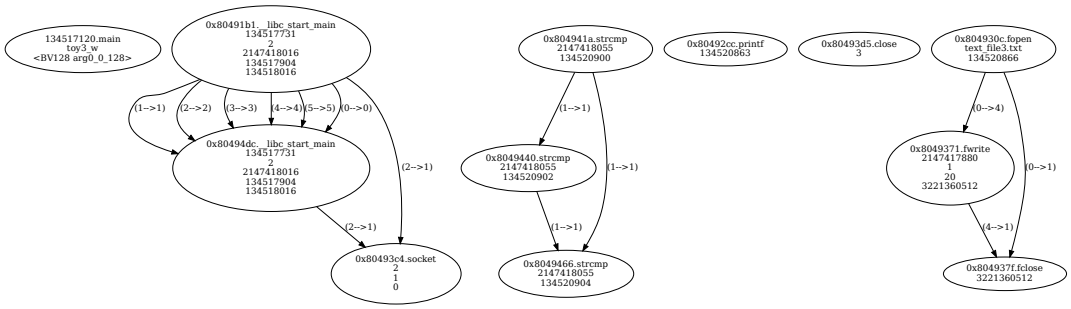


Figure 5.1: SCDG of the original binary.

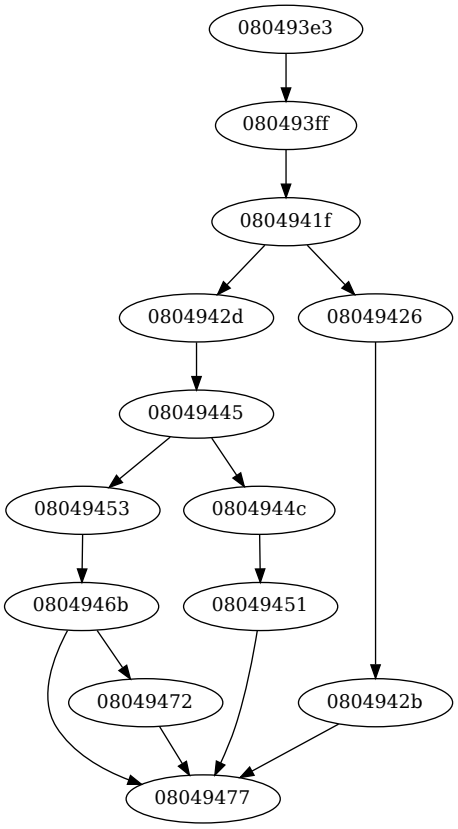


Figure 5.2: Code flow graph from Ghidra on the original binary.

5.2 Mutations effects

5.2.1 Strings

To analyze the string mutations, we will mainly use Yara. Since the only goal of those mutations is to obfuscate the presence of the strings, the SCDG is not relevant. Ghidra integrates tools for string reverse engineering, we will also take a look at that. We created different Yara rules focused on our toy example, here trying to detect the "I am evil" string for demonstration.

Split

To try to catch the splitting, we tried 4 different levels of intensity to find the split "I am evil". But as we will see, the higher the level, the more likely are the false positives. The four rules can be found in Appendix C.3.1. The first one search for plain text, without any modification. The second split the string in words, separated from a maximum of 16 bytes (or characters). The third split each character, separated by at most 16 bytes. This one is already problematic for false positives, as any sentences with those letters in the right order would match the rule. For example, a program with only this main function would match.

```
1 printf("I invite vibrant joy and limitless love into my life!");
```

The fourth one is a desperate attempt to still match the malware, by verifying if all the letters of the string are in the binary, ignoring any order. This is becoming extremely problematic as it would match a huge quantity of cleanwares. For example:

```
1 printf("I beam with life, reveling in pure happiness!");
```

Let's now see how our mutation performs.

The original binary of course match all the rules:

```
1         ORIGINAL:
2 basic_string s2e/projects/eval/binary
3 splitted s2e/projects/eval/binary
4 extreme_split s2e/projects/eval/binary
5 all_letters s2e/projects/eval/binary
```

The `split --text` command matches two rules. In fact, as discussed in Section 3.1.4, the string is only split because of the instructions between the parts of the string. Therefore, no matter what options we will try, it will always be matched by `extreme_split` and `all_letters`.

```
        MUTATED:
extreme_split s2e/projects/eval/s2e-out/custom_recovered
all_letters s2e/projects/eval/s2e-out/custom_recovered
```

When applying the `split`, we now only match the `all_letters`, which is extremely good as it means the only way to find the string is to randomly checks the right letters. This mutation is equivalent to a shuffle of the letters (because we split each letter and then store them in random order). If we want to deactivate this shuffle to see if `extreme_split` can match the string, then suddenly we match all rules event the `basic_string` as all the parts are stored in order and next to each other, we achieve nothing without shuffling.

```
        MUTATED:
all_letters s2e/projects/eval/s2e-out/custom_recovered
```

XOR

The Yara XOR modifier seems at first glance the perfect modifier to catch this mutation (Appendix C.3.2). But despite trying all the different flavors it proposes, the rule never matches our mutant ...

This is explained by how we execute our XOR:

```
; Replace: store i32 134520840, i32* %53, align 8
@str.7 = private unnamed_addr constant [14 x i8]
 ↪  c"\1B\48\22\54\13\08\42\0E\26\51\14\51\48\61", align 1
@key.7 = private unnamed_addr constant [14 x i8]
 ↪  c"\52\68\43\39\33\6d\34\67\4a\70\35\70\42\61", align 1
```

`@str.7` is the xored string and `@key.7` is the key. We can see that the key is the same length as the string, providing in fact a one-time pad encryption. There is a new random key for each string and, being random, new keys for each run of the mutation. So the same string will be encrypted the same way with a probability of

$$P_{(collision)} = 256^{length}$$

Which gives Yara no chance of finding the string.

The XOR Yara modifier works by testing all the 1-byte keys to xor with the characters. It is therefore incapable of finding our string, the only way it would, would be that the key is all the same byte, as shown below.

```
; Replace: store i32 134520840, i32* %53, align 8
@str.7 = private unnamed_addr constant [14 x i8]
 ↪  c"\e2\8b\ca\c6\8b\ce\dd\c2\c7\8a\8a\8a\a1\ab", align 1
@key.7 = private unnamed_addr constant [14 x i8]
 ↪  c"\ab\ab\ab\ab\ab\ab\ab\ab\ab\ab\ab\ab\ab\ab", align 1
```

```
        MUTATED:
XORed s2e/projects/eval/s2e-out/custom_recovered
all_letters s2e/projects/eval/s2e-out/custom_recovered
```

We also observe that the `all_letters` rule is so bad that it matches also in this case.

Base64

The base64 encryption goal was mainly to show that we can use external arbitrary C functions to decrypt the cipher. It is no surprise that the base64 mutant is matched by the base64 rule.

```
        MUTATED:
Base64 s2e/projects/eval/s2e-out/custom_recovered
all_letters s2e/projects/eval/s2e-out/custom_recovered
```

We could escape this detection by running a split afterward, shuffling all the base64 characters.

Using Ghidra

Using Ghidra to explore the strings, the results are the same for all three technics. The splitter, xored or encoded string does not appear in the string panel, but our obfuscation wouldn't last long against a reverse engineer. They are a lot of clues, such as traces of the use of `llvm-link`, the imported function names or

the names of the variables used in LLVM that could lead someone dedicated to understanding the basic inner workings of our string manipulations. An additional obfuscation layer would solve this problem, basically a randomization of the variable and function names.

5.2.2 Clean code Addition

For the Clean code addition, only the analysis of the SCDG is relevant. Indeed, as we add a call to the cleanware into the main function, the main code flow will not change. An important thing to remember is that the insertion of the cleanware happens at random places, and the chosen cleanwares are also picked at random, in a database of 232 cleanware. We will analyze a trace generated using the following command :

```
1 just mutate to_mutate clean_adder --number_add 18
```

Using the toolchain, we used the produced executable to create a graph, and we had the following results:

- <SimulationManager with 4 deadended>
- Total number of blocks: 299
- Total number of instr: 2988
- Number of blocks visited: 291
- Number of instr visited: 2406
- Syscalls Found:'malloc': 13, '___libc_start_main': 6, 'strlen': 174, 'printf': 139, 'putchar': 33, 'puts': 12, 'strncmp': 3, 'strcmp': 3, '___ctype_b_loc': 1, 'fopen': 1, 'tmpfile': 1, 'fwrite': 2, 'free': 2, 'fseek': 1, 'rewind': 1, 'fread': 1, 'close': 3, 'fclose': 2, 'socket': 1, 'access': 1
- Number of nodes: 97
- Number of links: 197

Because we added 18 random cleanwares, it makes sense to have these numbers: we added code, so we have more blocks, more instructions, and about the same average of blocks and instructions have been visited compared to the original. The cleanwares have many systems calls in them, though it is often `strlen`, explaining the higher number of nodes and links.

The same executable was used to create the graph available in C.2.1. In order to make it more understandable and readable, we only took here the chunks that interest us (Figure 5.3).

Not represented on this chunk of the graph is the consequent amount of system calls that are on their own. This is because the cleanware are not linked to one another, or linked with the initial main function.

Though cleanware are technically not linked to one another, some addresses of the original code of the cleanware remain. These addresses prove to be a match with useful data and allow thus a link between the cleanware and the initial project. This completely drowns the original calls among the rest, and even the original `strcmp` functions are anecdotic among everything.
But this cleanware is a good case scenario. There might be times when the cleanware stays by itself and does not link with the rest, but this all depends on the amount of cleanware added, which cleanware are chosen, and where they are used.

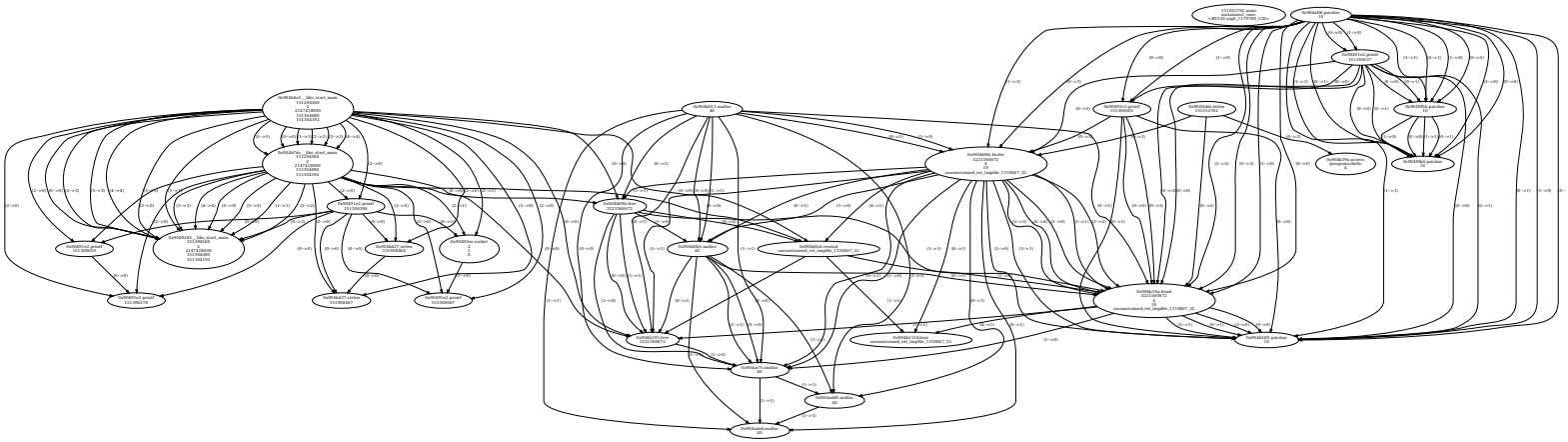


Figure 5.3: SCDG after adding 19 random cleanware

5.2.3 Workflow Hacking

Basic if

For the basic if, we will analyze the SCDG and the Ghidra code flow graph. So after running the command

```
just mutate to_mutate basic_if --words troisieme okiojvi,er Premier
↪  troisieme okiojvi,er Premier Premier Premier Premier zefiĵht
↪  Deuxième troisieme
```

that adds 12 basic ifs, we first obtained the graph 5.4. Though we can still see the elements of the initial SCDG, we can see the extra `puts` implemented by the mutation, drowning the original put in the SCDG 5.4. Depending on what they print, these `puts` are linked or not together. on the graph on the very left, we can see all the links created by the conditions. The toolchain gave us this information about the mutation :

- <SimulationManager with 4 deadended>
- Total number of blocks: 82
- Total number of instr: 443
- Number of blocks visited: 76
- Number of instr visited: 341
- Syscalls Found:'malloc': 8, ' ___libc_start_main': 6, 'strlen': 22, 'puts': 12, 'strncmp': 3, 'strcmp': 3, 'printf': 1, 'fopen': 1, 'fwrite': 1, 'socket': 1, 'close': 2, 'fclose': 1
- Number of nodes: 25
- Number of links: 55

We can see that just a few blocks and a few lines have been added. This makes sense since the only thing we have done here is to add 12 basic if conditions.

Then, we generated the graph 5.5 from Ghidra. The graph is similar to the original one, but the four original branches are harder to identify. Their lengths have changed, and the addition of branches change their original control flow. Each time there is a branch, the one that has an extra code block can be identified as the extra branch that prints a variable.
We do not think that this mutation is useful by itself. Indeed, we can still identify

every main component that identifies the original binary: the system calls and the branches.

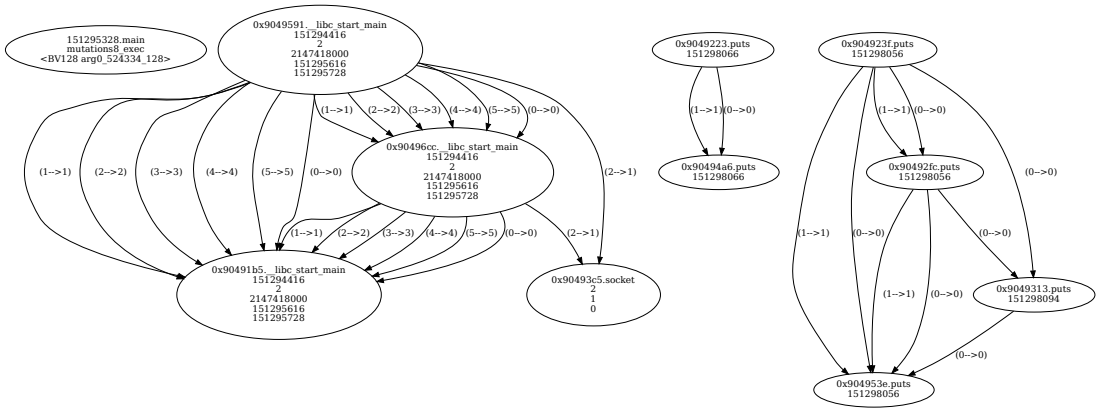


Figure 5.4: SCDG after using the basic if mutation.

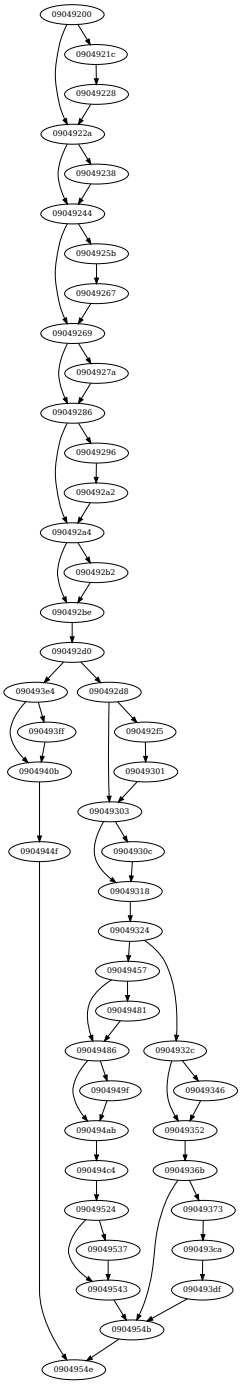


Figure 5.5: Code flow graph from Ghidra after using the basic if mutation.

Random if

For the random if mutator, we will analyze the SCDG and the graph provided from Ghidra. To obtain this graph, we ran the following command:

```
just mutate to_mutate random_if --max_random 11 --number 10
```

. This command adds 10 random if's, and each has a 1/4 probability to succeed. On the graph 5.6, we can see that the `strcmp` nodes haven't completely vanished from the graph, but they are linked to the main block anymore. Instead, we see all the added conditions, and where they return to. The bigger graph represents the 10 added conditions. The `rand` calls are all linked to one another, are linked because they all use `time` to compute their value. Let's analyze the result from the toolchain :

- <SimulationManager with 14 deadended>
- Total number of blocks: 81
- Total number of instr: 543
- Number of blocks visited: 77
- Number of instr visited: 436
- Syscalls Found:'malloc': 8, '__libc_start_main': 16, 'time': 2, 'srand': 1, 'rand': 10, 'strlen': 10, 'strncmp': 3, 'strcmp': 3, 'printf': 1, 'fopen': 1, 'socket': 1, 'fwrite': 1, 'close': 2, 'fclose': 1
- Number of nodes: 25
- Number of links: 84

The analysis of these results is the same as for the previous mutation. However, we can see that we now have 14 dead ends, caused by the 10 if conditions that we added, and that exit the program when the condition is false.

The graph 5.7 looks like the original one, but has many more branches. This is due because of the conditions we added. This makes the original three conditions harder to identify, though we can still guess which they are based on the length of the branches and their content.
This mutation is thus useful in the case of an SCDG analysis but needs to be paired with another mutation to completely hide the original program.

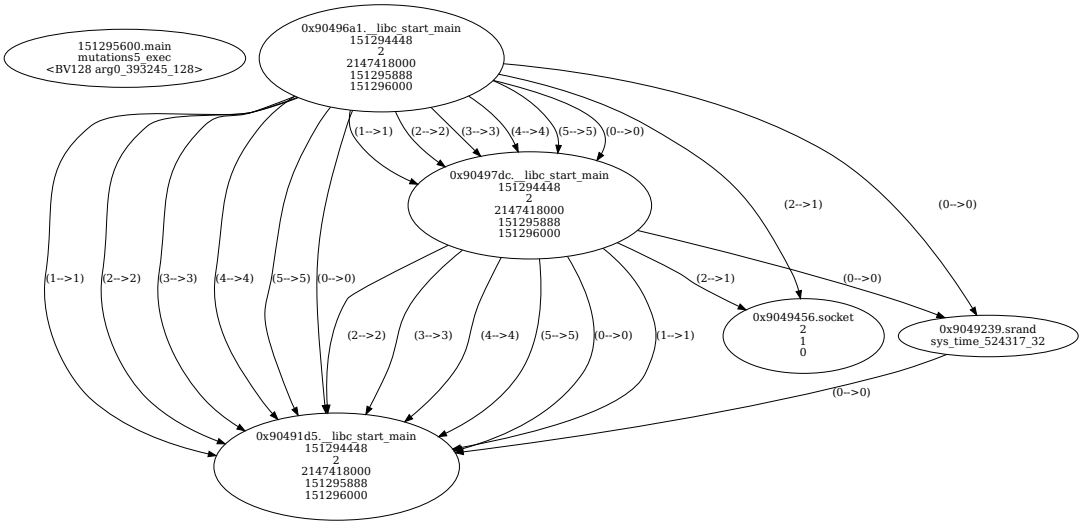


Figure 5.6: SCDG after using the random if mutation.

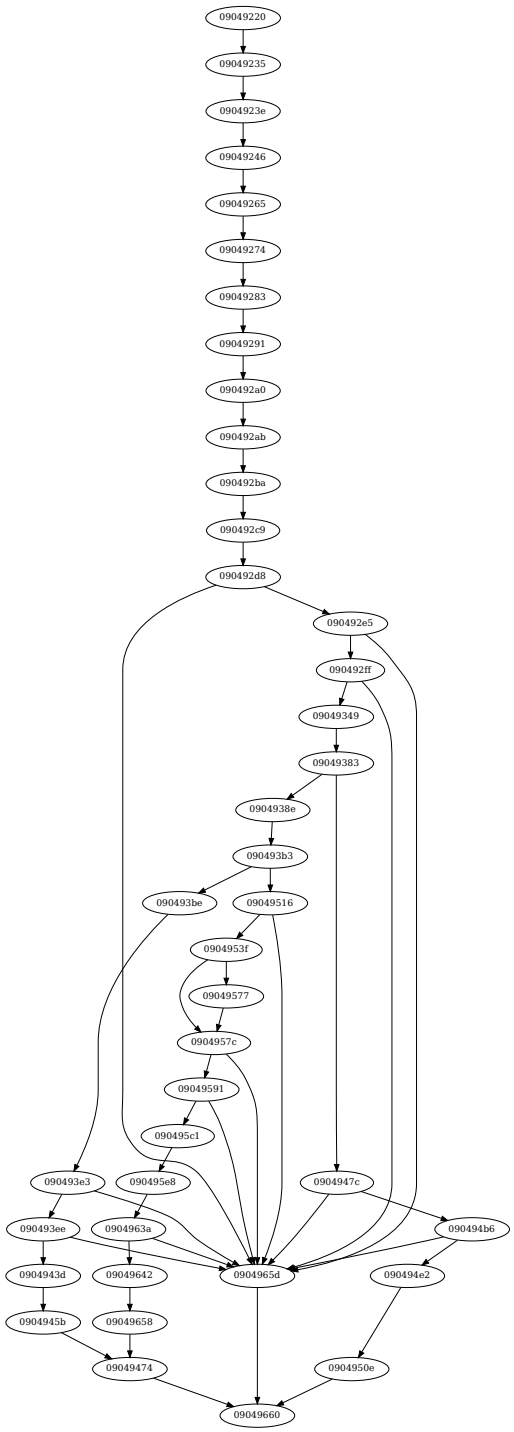


Figure 5.7: Code flow graph from Ghidra after using the random if mutation.

5.2.4 API Call Addition

Just like for the cleanware addition, it is only pertinent to look at the SCDG. After mutating the original project with this command

```
1 just mutate to_mutate sys_adder --number_add 18
```

we obtained the SCDG 5.8 and the following data:

- <SimulationManager with 3 deadended (1 errored)>
- Total number of blocks: 75
- Total number of instr: 433
- Number of blocks visited: 68
- Number of instr visited: 278
- Syscalls Found:'malloc': 8, '___libc_start_main': 5, 'strlen': 20, 'strncmp': 4, 'strcmp': 4, 'strstr': 1, 'access': 1, 'toupper': 1, 'tolower': 1, 'strchr': 1, 'puts': 1, 'htonl': 1, 'fopen': 1, 'socket': 1, 'alarm': 1, 'atof': 1, 'close': 2, 'time': 2, 'fwrite': 1, 'getenv': 1, 'atol': 1, 'fclose': 1
- Number of nodes: 27
- Number of links: 43

We can see two important things: first, we see that we only have three deadends but also one error. As our program executes without issues, we think that this error comes from the symbolic execution of our mutant. It would be caused by one of the added API calls for which the simProcedure (a summary of the call characteristics) is not implemented in the toolchain.
Second, we see that 11 system calls have been added to the original project. We only have 11 new system calls because some were already in the project, and also because of the error, some calls were not visited and thus not registered. There are fewer new links in this workflow mutation because the added calls are only linked to one variable.
As we expected, we can see in the graph that many system calls have been mixed with the original calls, drowning them in the mass. A few calls are not linked to the main block, and this can be explained by the fact that the variables we use are potential variables. If the variable called is not used by another system call, it will not be linked to the rest.

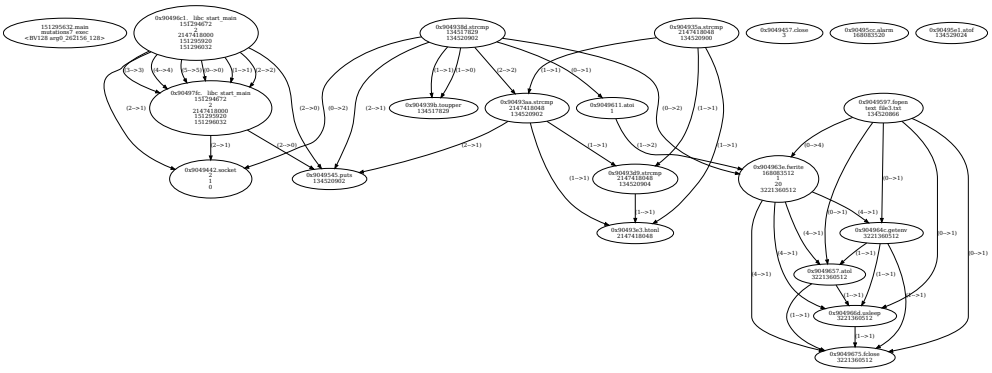


Figure 5.8: SCDG after using the system call adder mutation.

5.2.5 Escapes

All the following mutations are a demonstration of different techniques to evade specific environments, they could be extended far beyond what we've done.

Environment Variables

In our tests, we used a list of debug variables, composed of DEBUG, GDB, PYTHONDEBUG, LD_DEBUG, VBOX_USER_HOME, VMWARE_DEBUG, HYPERV_BOOT_DEBUG. The escape is effective, we observe here that the mutant shutdown when the GDB variable is set.

```
$ echo $GDB

$ just exec-mutated eval 1
"s2e/projects/eval/s2e-out/custom_recovered" 1
I am evil!!!
$ export GDB=1
$ just exec-mutated eval 1
"s2e/projects/eval/s2e-out/custom_recovered" 1
$ unset GDB
$ just exec-mutated eval 1
"s2e/projects/eval/s2e-out/custom_recovered" 1
I am evil!!!
$
```

Tracing detection

As the tracing is testing if `ptrace` can establish a link with the process, this could not be a universal escape, but it worked in our tests with *strace*, *gdb* and *lldb*. Here are the ends of the *strace* command on the original binary and the mutant.

```
mprotect(0xf7ec1000, 8192, PROT_READ)   = 0
mprotect(0x904b000, 4096, PROT_READ)    = 0
mprotect(0xf7f13000, 4096, PROT_READ)   = 0
munmap(0xf7ec7000, 100531)              = 0
fstat64(1, {st_mode=S_IFCHR|0620,
↪  st_rdev=makedev(0x88, 0x4), ...}) = 0
brk(NULL)                               =
↪  0xbe54000
brk(0xbe75000)                          =
↪  0xbe75000
brk(0xbe76000)                          =
↪  0xbe76000
write(1, "I am evil!!!\n", 13I am evil!!!
)          = 13
exit_group(0)                           = ?
+++ exited with 0 +++
```

```
mprotect(0xf7f75000, 8192, PROT_READ)   = 0
mprotect(0x904b000, 4096, PROT_READ)    = 0
mprotect(0xf7fc7000, 4096, PROT_READ)   = 0
munmap(0xf7f7b000, 100531)              = 0
ptrace(PTRACE_TRACEME)                  = -1
↪  EPERM (Opération non permise)
exit_group(0)                           = ?
+++ exited with 0 +++
```

Of course, running the mutant outside of the *strace* gives us the usual behavior.

```
$ ./s2e/projects/eval/s2e-out/custom_recovered 1
I am evil!!!
```

VM Detection

The code we used to detect a VM environment only works for VMWare, but it is a good proof of work and could easily be extended to any detection technique. Running the mutant, as usual, does not change anything.

```
$ s2e/projects/eval/s2e-out/custom_recovered 1
Command not found or exited with error status
I am evil!!!
```

But inside a vm, it becomes silent.

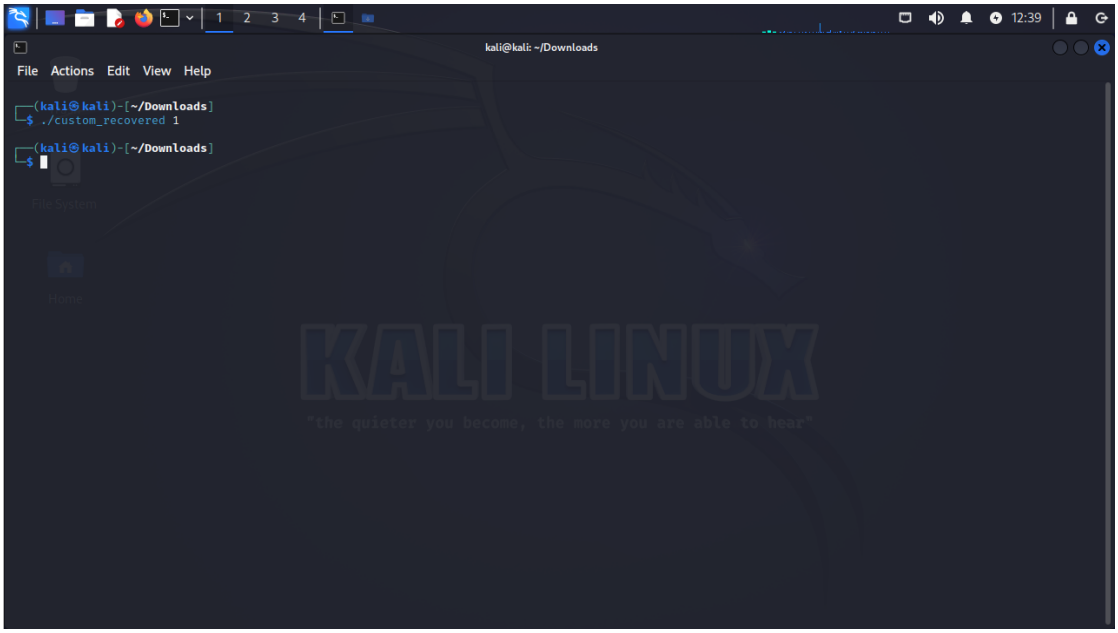


Figure 5.9: The mutant running inside the VMWare environment

For this test, we used the standard *kali-linux-2023.1-vmware-i386* VMWare image distributed on the Kali Linux website.

5.2.6 Puts Replacement

As this mutation is destined to be a proof of work, another project was used. First, we created a basic C program available at C.1.1. To mutate the project, we used the following command :

```
1 just mutate puts replace_puts
```

To analyze the results, we used a different tool in Ghidra than previously. To prove that the change happened, we will look at the function call trees before the mutation 5.10 and after the mutation 5.11. As we can see, the outgoing reference to `puts` has been successfully replaced by a reference to `printf`

Figure 5.10: Outgoing calls before mutation

Figure 5.11: Outgoing calls after mutation

5.3 Analysis of a Complete Mutation

To conclude this section, we will analyze a mutation that used every tool we presented, except for the three "escape" mutations. We could not use these because of the limitations relative to the toolchain. We were not able to retrieve an SCDG when using these mutations.
We generated the SCDG and the code flow using the following commands that we generated at random:

```
1 just mutate to_mutate random_if --max_random 17 --number 4
2 just mutate to_mutate strings xor --ncuts -1
3 just mutate to_mutate clean_adder --number_add 201
4 just mutate to_mutate strings base64 --ncuts -1
5 just mutate to_mutate sys_adder --number_add 18
6 just mutate to_mutate replace_puts
7 just mutate to_mutate strings split --ncuts 13
```

- <SimulationManager with all stashes empty (2 errored)>
- Total number of blocks: 3838

- Total number of instr: 49607
- Number of blocks visited: 281
- Number of instr visited: 2574
- Syscalls Found:'malloc': 82, '__libc_start_main': 2, 'time': 2, 'srand': 1, 'strlen': 159, 'printf': 78, 'rand': 4, 'putchar': 6, 'tmpfile': 1, 'fwrite': 1, 'free': 2, 'fseek': 1, 'rewind': 1, 'fread': 1, 'close': 1, 'fclose': 1, 'usleep': 1, 'strtok': 1, 'puts': 1, 'tolower': 22
- Number of nodes: 78
- Number of links: 557

The result from the toolchain shows us that we have 2 errors. However, just like for the system call adder, our executable reaches the end of the program normally. We think that again, this is due to the symbolic execution of the program. This can influence the SCDG, but we can still see the main components that interest us. As expected, we have much more blocks and instructions because of the extra cleanwares, but few are visited because of the errors. We expect that without errors, we would have timed out, and the number of blocks and instructions visited, the number of system calls found, and the number of nodes and links would be much higher.

The complete graphs are available in Appendix C.5 and Appendix **??**

First, let's analyze the SCDG. Despite the errors we talked about, we can still retrieve useful information. We see that the SCDG comprises three main graphs, which we will develop. The biggest one, Figure 5.12, is linked to the initial main that we can see in the bottom left corner. Then, it explodes into many links to different variables. This is due to the cleanwares that are added. As we mentioned earlier, links are likely made because of addresses that are present in both the original file and the different cleanwares.
The second block, figure 5.13, between the two other graphs, represents the random if mutations that were added. We can also see the impact of the replace puts mutation, as there is not a single `puts` left in the graph, but many `printf` can be seen. The toolchain still detects one `puts`, and it is the puts that we added with the system call adder mutation.
The last block, figure 5.14, is a similar scenario to the first block. All the functions are linked because they use variables common to some cleanwares.
The other mutations do not have a noticeable impact on the SCDG. Indeed, the string mutations do not have an impact at all, and the basic if mutation and the

system call adder mutation are not consequential enough to have an impact. They are likely lost among the biggest block, as they use system calls that are linked to the main project.

Second, we can take a look at the code flow graph generated from Ghidra on Figure 5.15. The splits can come from three different sources:

1. The basic if conditions
2. The random if conditions
3. Conditions already present in the cleanware.

As the number of random if mutations are not substantial, we do not have as many splits as we could have: more random if mutations would mean a graph with even more splits.

In conclusion, we can say that we have altered the original binary signature with success. The original behavior is unrecognizable in both graphs.

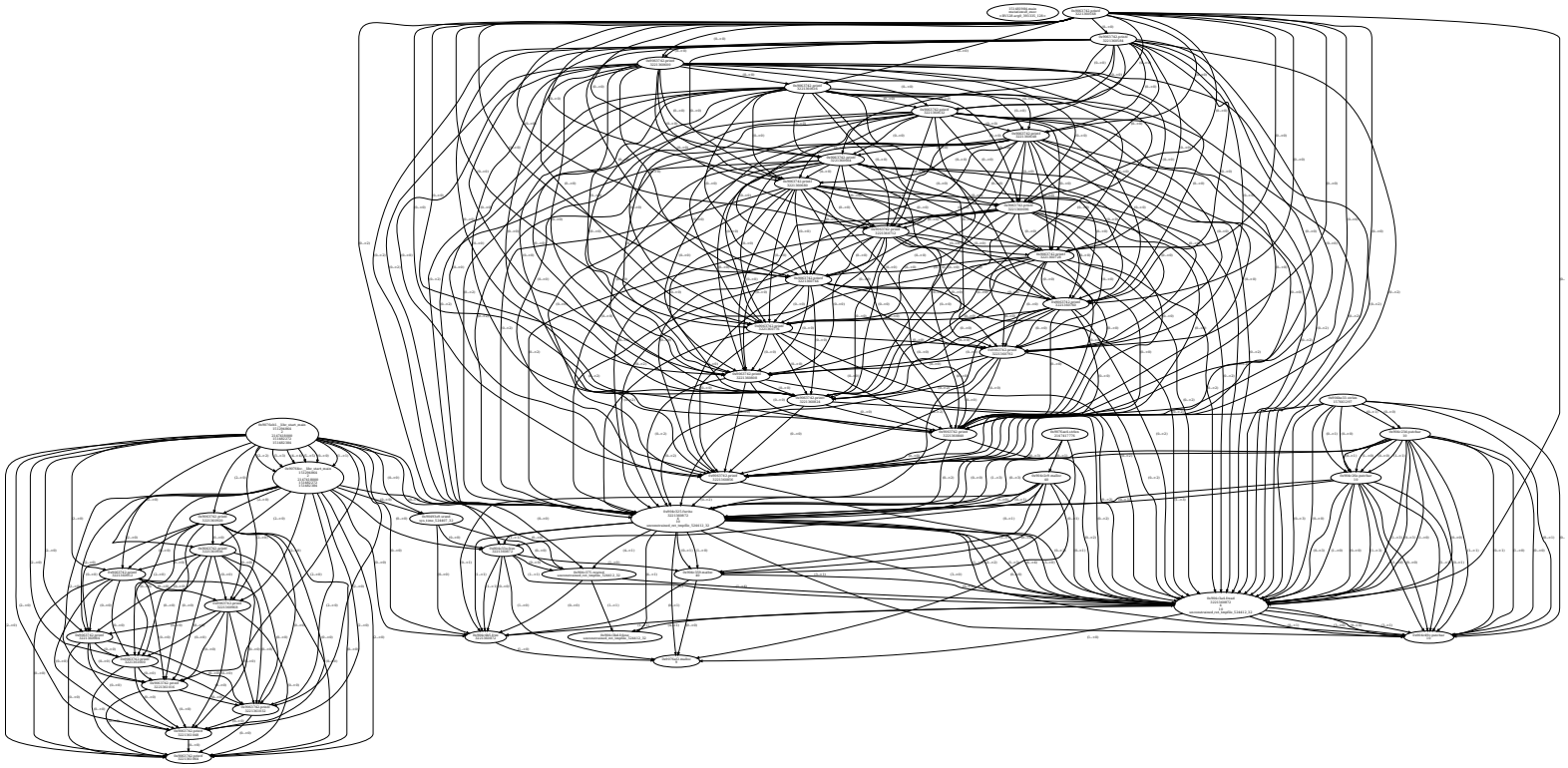


Figure 5.12: Main block of the extreme mutation

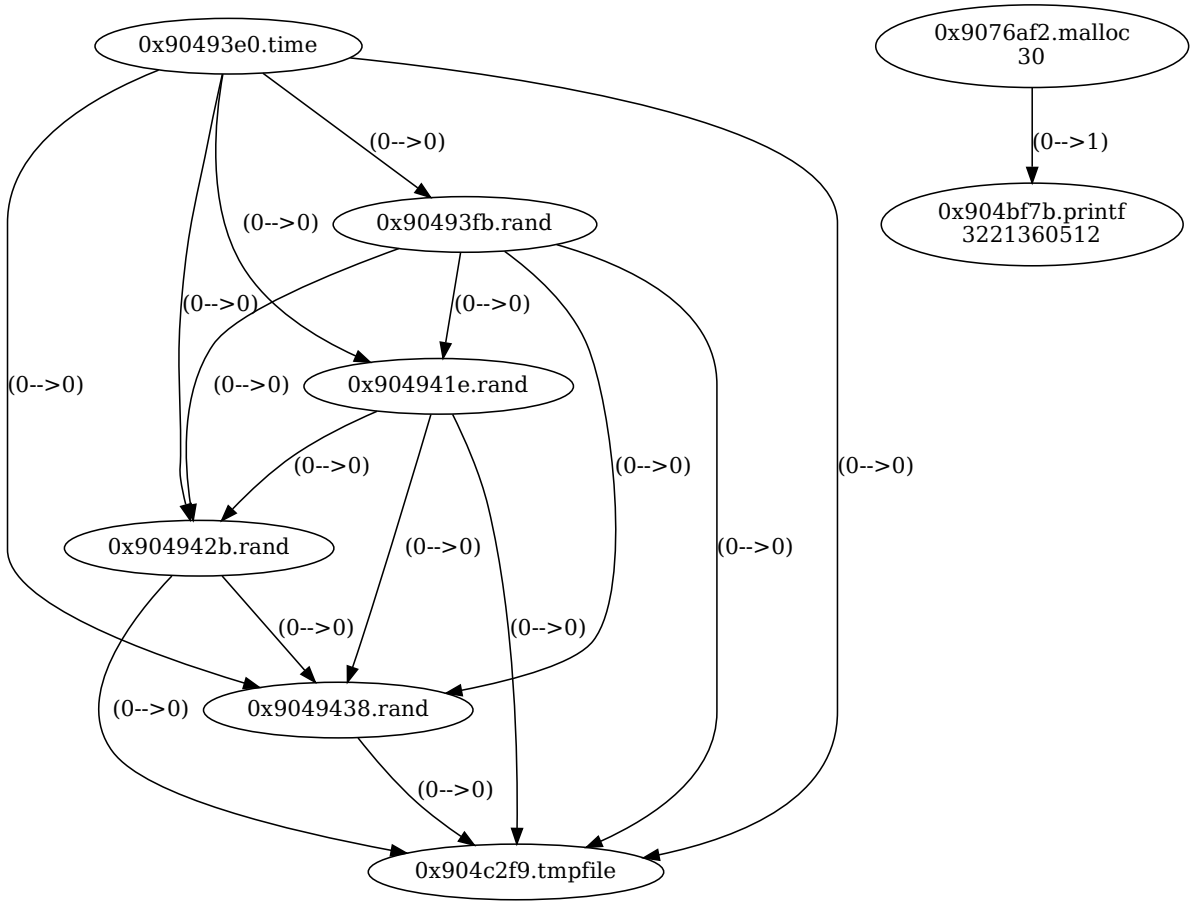


Figure 5.13: Random if block

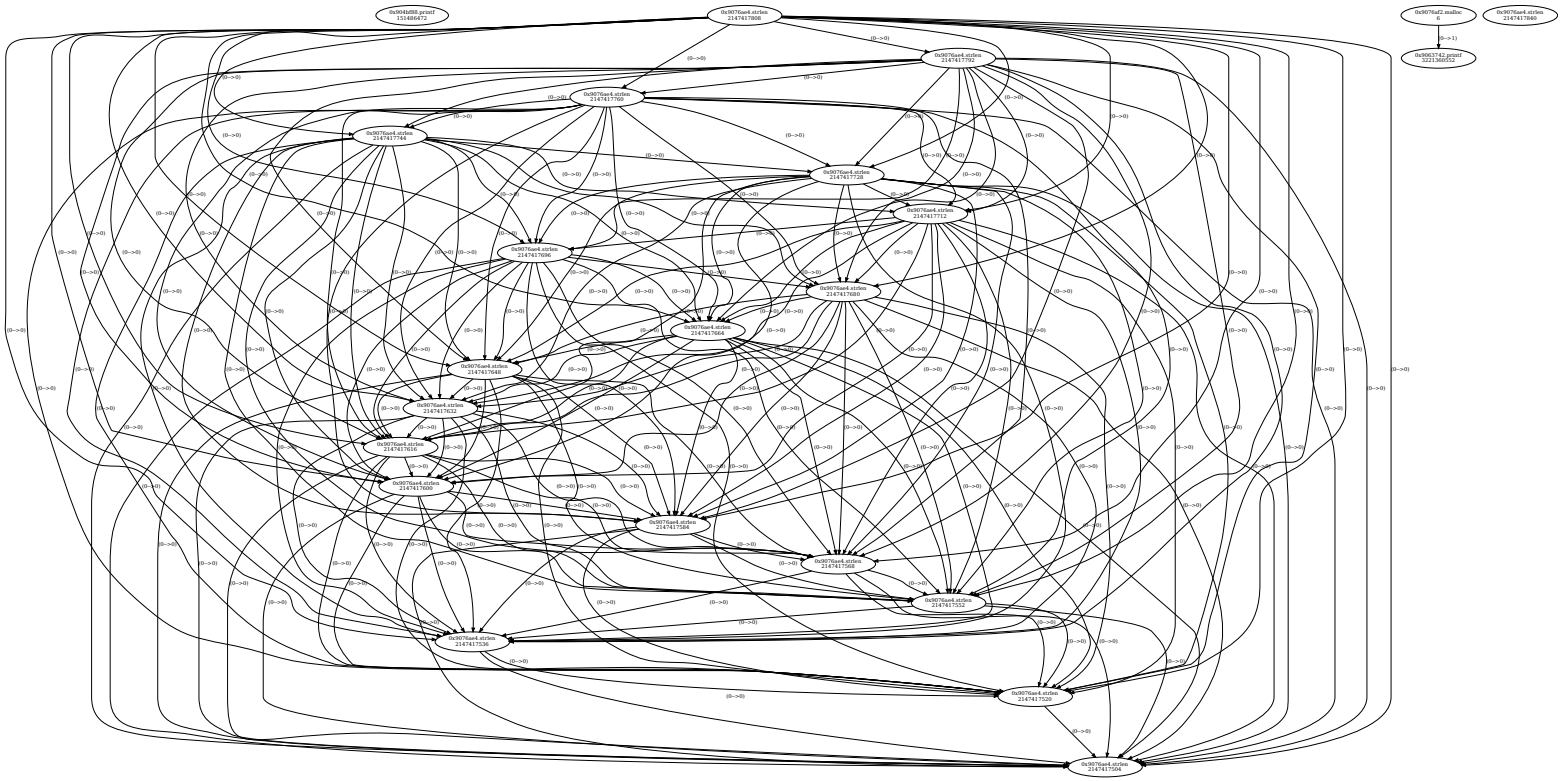


Figure 5.14: Last block, similar to the first one

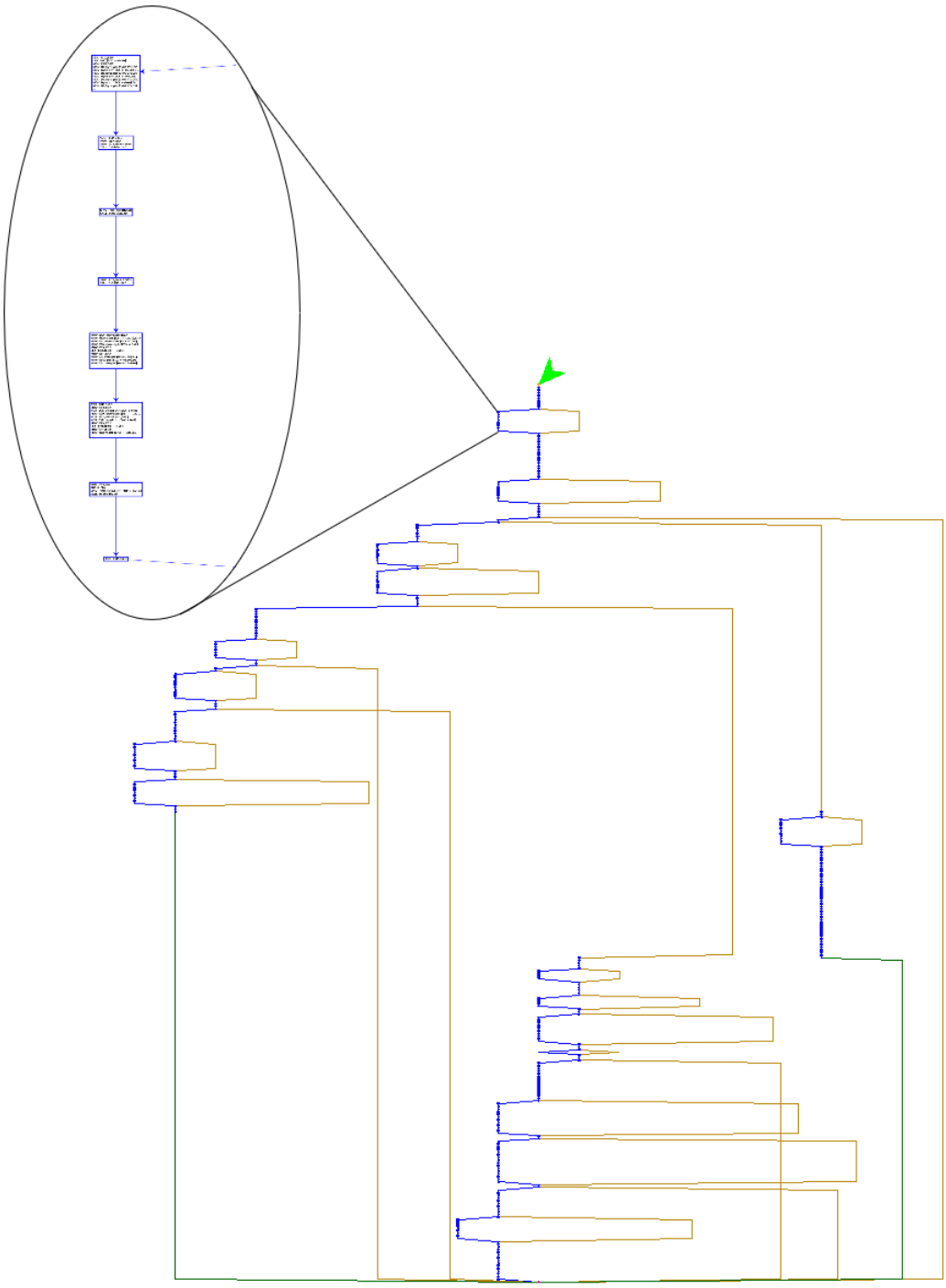


Figure 5.15: Code flow analyzed by Ghidra

Chapter 6

Future Work

The first thing we would advise working on is *BinRec*. *BinRec* can lift some binaries, but it is quite limited. Some functions are not correctly lifted, such as socket functions, which gave us trouble. Moreover, *BinRec* cannot successfully lift a binary if it is the result of multiple programs compiled in one executable. As malware executables are often a result of multiple program files compiled together, it greatly limits the use of our tool. Once *BinRec* is improved, we encourage the reader to test the existing tool first against malware classifiers before starting to work on the following steps.

Once that is done, we encourage the reader to try everything first one very simple malwares, and increase the complexity as time goes. We focused in this work on Linux malwares mutations, and research should also be done with Windows malwares, trying to find another tool to lift those files.

As we discussed in section 3.1.4, the .TEXT section is automatically optimized when recompiled. If a solution were to be found to avoid these optimizations, our manipulations on these sections would be ready for use.

Instead of splitting the strings and storing their values into the recovered file, we could go through the binary, find random bytes that have the value of one or multiple characters, and point to those bytes. This would further hide the strings, as there would not even be any of the string parts in the recovered file.

To further secure the encryption, another byte-encoding method could be used, other than the two already developed in this project, which would be even harder to break. Encryption schemes used in cryptography, that insure perfect secrecy, could be used.

We also recommend working on the replacement of other system calls. We

successfully proved that it was possible, and it would be interesting to keep working on it, by replacing functions such as `fwrite`, `execve` by their equivalent when it is possible. This replacement could throw off a classifier that classifies using the system calls found inside malware.

Another possibility of amelioration concerns the cleanware adder. Instead of just calling them randomly, it would be interesting to find some cleanware that takes input, to link the cleanware even more with the rest of the project. We expect that this would have an impact on the resulting SCDG.

Lastly, we could add an obfuscation layer to prevent our mutations to be reversed by a pre-treatment of the binary. LLVM already provides an obfuscation option, but tests are needed to make sure that it provides enough to protect our mutations. If it is not, we advise implementing the layer.

Chapter 7

Conclusion

The purpose of this work was to create a mutation tool for binaries. The interest of this research was to help malware classifiers correctly classify unknown malware that are mutations of previously known malware.

With this work, we confirm that it is totally possible to mutate a binary without any prior knowledge of its source code and without any human intervention.
For this, we developed eleven mutations that can be applied to binaries satisfying *BinRec*'s requirements. Three of them are directly applicable to the strings within a binary: a mutation split the strings and shuffles them, the second one xor the strings and the last one converts them into base64.
Another three mutations are directed towards evasion of different environments: one avoids virtual machines, though it needs to be tweaked, the second exits if its process is traced, and the last one exit if it runs in an environment with certain variables set. Four mutations add code to the project: the first adds system calls, and another adds cleanware that was previously generated; also, one adds basic if conditions that are always true, and the last one inserts random if conditions, that exit the program if they are false.
Finally, the last mutation allows us to change a `puts` with a `printf`, successfully proving that replacing system calls with other ones can be done.

We have met many challenges, the biggest one being the limitations of *BinRec*. We hope that in the not-so-distant future, a version of *BinRec* capable of lifting malware will emerge, thus allowing a dynamic mutation of malware.

Nonetheless, as mentioned earlier, we have succeeded in automating several mutations, proving that mutation can be done on a large scale, and opens the way for a later automation of malware mutation to help classifiers.

Bibliography

[1] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. "BinRec: dynamic binary lifting and recompilation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[2] Charles-Henri Bertrand Van Ouytsel, Christophe Crochet, Khanh Huu The Dam, Serena Lucca, and Axel Legay. *SEMA-Toolchain*. `https://github.com/csvl/SEMA-ToolChain`. 2023.

[3] Charles-Henry Bertrand Van Ouytsel, Christophe Crochet, and Axel Legay. "Tool paper-SEMA: Symbolic Execution toolchain for Malware Analysis". In: *17th International Conference on Risks and Security of Internet and Systems*. 2022.

[4] Charles-Henry Bertrand Van Ouytsel and Axel Legay. "Malware Analysis with Symbolic Execution and Graph Kernel". In: *Secure IT Systems: 27th Nordic Conference, NordSec 2022, Reykjavic, Iceland, November 30–December 2, 2022, Proceedings*. Springer. 2023, pp. 292–310.

[5] Fabrizio Biondi, Thomas Given-Wilson, Axel Legay, Cassius Puodzius, and Jean Quilbeuf. "Tutorial: An overview of malware detection and evasion techniques". In: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I 8*. Springer. 2018, pp. 565–586.

[6] Andrea Cani, Marco Gaudesi, Ernesto Sanchez, Giovanni Squillero, and Alberto Tonda. "Towards automated malware creation: code generation and code integration". In: *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 2014, pp. 157–160.

[7] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo. "Aimed: Evolving malware with genetic programming to evade detection". In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE. 2019, pp. 240–247.

[8] Fabio De Gaspari, Dorjan Hitaj, Giulio Pagnotta, Lorenzo De Carli, and Luigi V Mancini. "The naked sun: Malicious cooperation between benign-looking processes". In: *Applied Cryptography and Network Security: 18th International Conference, ACNS 2020, Rome, Italy, October 19–22, 2020, Proceedings, Part II 18*. Springer. 2020, pp. 254–274.

[9] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. "rev.ng: a unified binary analysis framework to recover CFGs and function boundaries". In: *Proceedings of the 26th International Conference on Compiler Construction*. 2017, pp. 131–141.

[10] Artem Dinaburg and Andrew Ruef. "Mcsema: Static translation of x86 instructions to llvm". In: *ReCon 2014 Conference, Montreal, Canada*. 2014.

[11] NSA's Research Directorate. *Ghidra*. `https://ghidra-sre.org/`. [Online; accessed 29-May-2023].

[12] Fenil Fadadu, Anand Handa, Nitesh Kumar, and Sandeep Kumar Shukla. "Evading API call sequence based malware classifiers". In: *Information and Communications Security: 21st International Conference, ICICS 2019, Beijing, China, December 15–17, 2019, Revised Selected Papers 21*. Springer. 2020, pp. 18–33.

[13] Victor Manuel Alvarez – Google. *YARA - The pattern matching swiss knife for malware researchers*. `https://virustotal.github.io/yara/`. [Online; accessed 29-May-2023].

[14] gouravthakur39. *beginners-C-program-examples*. `https://github.com/gouravthakur39/beginners-C-program-examples.git`. 2022.

[15] LLVM Compiler Infrastructure. *LLVM Command Guide*. `https://llvm.org/docs/CommandGuide/index.html`. Accessed: 2023-05-23.

[16] LLVM Compiler Infrastructure. *LLVM Language Reference Manual*. `https://llvm.org/docs/LangRef.html`. Accessed: 2023-05-23.

[17] Kyriakos K Ispoglou and Mathias Payer. "{malWASH}: Washing Malware to Evade Dynamic Analysis". In: *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. 2016.

[18] JakenHerman. *vm-detection*. `https://github.com/JakenHerman/vm-detection`. 2023.

[19] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. "Automating mimicry attacks using static binary analysis". In: *USENIX Security Symposium*. Vol. 14. 2005, pp. 11–11.

[20] Yunus Kucuk and Guanhua Yan. “Deceiving portable executable malware classifiers into targeted misclassification with practical adversarial examples”. In: *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*. 2020, pp. 341–352.

[21] Weiqin Ma, Pu Duan, Sanmin Liu, Guofei Gu, and Jyh-Charn Liu. “Shadow attacks: automatically evading system-call-behavior based malware detection”. In: *Journal in Computer Virology* 8 (2012), pp. 1–13.

[22] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. “Dynamic malware analysis in the modern era—A state of the art survey”. In: *ACM Computing Surveys (CSUR)* 52.5 (2019), pp. 1–48.

[23] Chetan Parampalli, R Sekar, and Rob Johnson. “A practical mimicry attack against powerful system-call monitors”. In: *Proceedings of the 2008 ACM symposium on Information, computer and communications security*. 2008, pp. 156–167.

[24] Jithin Pavithran, Milan Patnaik, and Chester Rebeiro. “D-TIME: Distributed Threadless Independent Malware Execution for Runtime Obfuscation.” In: *WOOT@ USENIX Security Symposium*. 2019.

[25] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. “Generic black-box end-to-end attack against state of the art API call based malware classifiers”. In: *Research in Attacks, Intrusions, and Defenses: 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings 21*. Springer. 2018, pp. 490–510.

[26] Ali Shafiei, Vera Rimmer, Ilias Tsingenopoulos, Lieven Desmet, and Wouter Joosen. “Position Paper: On Advancing Adversarial Malware Generation Using Dynamic Features”. In: *Proceedings of the 1st Workshop on Robust Malware Analysis*. 2022, pp. 15–20.

[27] National Institute of Standards and Technology. *Information Security*. Tech. rep. Information Security Continuous Monitoring (ISCM) for Federal Information Systems and Organizations 800-137. Washington, D.C.: U.S. Department of Commerce, 2011.

[28] Rabia Tahir. “A study on malware and malware detection techniques”. In: *International Journal of Education and Management Engineering* 8.2 (2018), p. 20.

[29] Sajedul Talukder. “Tools and techniques for malware detection and analysis”. In: *arXiv preprint arXiv:2002.06819* (2020).

[30] trailofbits. *binrec-tob*. `https://github.com/trailofbits/binrec-tob`. 2023.

[31] Charles-Henry Bertrand Van Ouytsel and Axel Legay. “Detection and classification of malware based on symbolic execution and machine learning methods”. In: *Cybersec & AI* (2021).

[32] Wikipedia. *Dynamic program analysis — Wikipedia, The Free Encyclopedia*. [Online; accessed 05-April-2023]. 2023. URL: https://en.wikipedia.org/wiki/Dynamic%5C_program%5C_analysis.

[33] Wikipedia. *Function prototype — Wikipedia, The Free Encyclopedia*. [Online; accessed 04-April-2023]. 2023. URL: http://en.wikipedia.org/w/index.php?title=Function%5C%20prototype&oldid=1108831273.

[34] Wikipedia contributors. *Malware — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Malware&oldid=1154749867. [Online; accessed 18-April-2023]. 2023.

[35] Wikipedia contributors. *Symbolic execution — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Symbolic_execution&oldid=1153725231. [Online; accessed 25-May-2023]. 2023.

[36] Wikipedia contributors. *Trampoline (computing) — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Trampoline_(computing)&oldid=1156569284. [Online; accessed 3-June-2023]. 2023.

[37] S Bharadwaj Yadavalli and Aaron Smith. “Raising binaries to LLVM IR with MCTOLL (WIP paper)”. In: *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 2019, pp. 213–218.

Appendix A

Mutations

A.1 Original lifted Hello World program

```
    ; ModuleID = 'optimized.bc'
source_filename = "llvm-link"
target datalayout = "e-m:e-p:32:32-p270:32:32-p271:32:32-p272:64:64-f64:32:64-f80:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

@fpstt = internal unnamed_addr global i32 0
@stack = internal global [4194304 x i32] zeroinitializer, align 16
@onUnfallback = common local_unnamed_addr global i1 false

; Function Attrs: mustprogress nofree norecurse nosync nounwind uwtable willreturn writeonly
define internal fastcc void @helper_fninit() unnamed_addr #0 {
  store i32 0, i32* @fpstt, align 16
  ret void
}

; Function Attrs: norecurse nounwind uwtable
define dso_local noundef i32 @main(i32 noundef %0, i8** noundef %1, i8** noundef %2) local_unnamed_addr #1 {
  %4 = tail call i32 asm "pushf\0A\09popl $0", "=r,~{dirflag},~{fpsr},~{flags}"() #5
```

```
19    tail call fastcc void @helper_fninit() #6
20    %5 = ptrtoint i8** %2 to i32
21    store i32 %5, i32* getelementptr inbounds ([4194304 x i32], [4194304
      ↪  x i32]* @stack, i32 0, i32 4194303), align 4, !tbaa !6
22    %6 = ptrtoint i8** %1 to i32
23    store i32 %6, i32* getelementptr inbounds ([4194304 x i32], [4194304
      ↪  x i32]* @stack, i32 0, i32 4194302), align 8, !tbaa !6
24    store i32 %0, i32* getelementptr inbounds ([4194304 x i32], [4194304
      ↪  x i32]* @stack, i32 0, i32 4194301), align 4, !tbaa !6
25    %7 = tail call i8* @llvm.returnaddress(i32 0)
26    %8 = ptrtoint i8* %7 to i32
27    store i32 %8, i32* getelementptr inbounds ([4194304 x i32], [4194304
      ↪  x i32]* @stack, i32 0, i32 4194300), align 16, !tbaa !6
28    tail call fastcc void @Func_main(i32 ptrtoint (i32* getelementptr
      ↪  inbounds ([4194304 x i32], [4194304 x i32]* @stack, i32 0, i32
      ↪  4194300) to i32)) #7
29    ret i32 0
30  }
31
32  ; Function Attrs: nofree nosync nounwind readnone willreturn
33  declare i8* @llvm.returnaddress(i32 immarg) #2
34
35  declare dso_local i32 @puts(i8* noundef) local_unnamed_addr #3
36
37  ; Function Attrs: norecurse
38  define internal fastcc void @Func_main(i32 %arg_esp) unnamed_addr #4
      ↪  !retregs !10 {
39    %tmp2_v.i.i = add i32 %arg_esp, 4
40    %tmp0_v.i.i = and i32 %arg_esp, -16
41    %1 = inttoptr i32 %arg_esp to i32*
42    %2 = load i32, i32* %1, align 4
43    %tmp2_v3.i.i = add i32 %tmp0_v.i.i, -4
44    %3 = inttoptr i32 %tmp2_v3.i.i to i32*
45    store i32 %2, i32* %3, align 4
46    %tmp2_v4.i.i = add i32 %tmp0_v.i.i, -8
```

```
  %4 = inttoptr i32 %tmp2_v4.i.i to i32*
  store i32 0, i32* %4, align 8
  %tmp2_v5.i.i = add i32 %tmp0_v.i.i, -12
  %5 = inttoptr i32 %tmp2_v5.i.i to i32*
  store i32 0, i32* %5, align 4
  %tmp2_v6.i.i = add i32 %tmp0_v.i.i, -16
  %6 = inttoptr i32 %tmp2_v6.i.i to i32*
  store i32 %tmp2_v.i.i, i32* %6, align 16
  %tmp2_v7.i.i = add i32 %tmp0_v.i.i, -20
  %7 = inttoptr i32 %tmp2_v7.i.i to i32*
  store i32 134517166, i32* %7, align 4
  %tmp2_v2.i.i = add i32 %tmp0_v.i.i, -32
  %8 = inttoptr i32 %tmp2_v2.i.i to i32*
  store i32 134520840, i32* %8, align 16
  %tmp2_v3.i11.i = add i32 %tmp0_v.i.i, -36
  %9 = inttoptr i32 %tmp2_v3.i11.i to i32*
  store i32 134517188, i32* %9, align 4
  %arg.i.i = load i32, i32* %8, align 16
  %10 = inttoptr i32 %arg.i.i to i8*
  %11 = tail call i32 @puts(i8* nonnull dereferenceable(1) %10)
  ret void
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind
↪  uwtable willreturn writeonly "frame-pointer"="all"
↪  "min-legal-vector-width"="0" "no-trapping-math"="true"
↪  "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" }
attributes #1 = { norecurse nounwind uwtable "frame-pointer"="none"
↪  "no-builtins" "no-trapping-math"="true"
↪  "stack-protector-buffer-size"="8" "target-cpu"="pentium4"
↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
↪  "tune-cpu"="generic" }
attributes #2 = { nofree nosync nounwind readnone willreturn }
```

```
73 attributes #3 = { "frame-pointer"="none" "no-builtins"
   ↪ "no-trapping-math"="true" "stack-protector-buffer-size"="8"
   ↪ "target-cpu"="pentium4"
   ↪ "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
   ↪ "tune-cpu"="generic" }
74 attributes #4 = { norecurse }
75 attributes #5 = { nounwind readnone }
76 attributes #6 = { nobuiltin nounwind "no-builtins" }
77 attributes #7 = { nounwind }
78
79 !llvm.ident = !{!0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0}
80 !llvm.module.flags = !{!1, !2, !3, !4, !5}
81
82 !0 = !{!"clang version 14.0.0"}
83 !1 = !{i32 1, !"wchar_size", i32 4}
84 !2 = !{i32 7, !"PIC Level", i32 2}
85 !3 = !{i32 7, !"uwtable", i32 1}
86 !4 = !{i32 7, !"frame-pointer", i32 2}
87 !5 = !{i32 1, !"NumRegisterParameters", i32 0}
88 !6 = !{!7, !7, i64 0}
89 !7 = !{!"int", !8, i64 0}
90 !8 = !{!"omnipotent char", !9, i64 0}
91 !9 = !{!"Simple C++ TBAA"}
92 !10 = !{i32 0, i32 0, i32 0}
```

Code A.1.1: LLVM code after lifting a "Hello World" program.

A.2 Generated LLVM string loader

```
1 %sp1 = alloca [13 x i8]
2
3 %s0.1 = load [1 x i8], [1 x i8]* @str.0.1
4 %sp0.1 = bitcast [13 x i8]* %sp1 to [1 x i8]*
```

```
store [1 x i8] %s0.1, [1 x i8]* %sp0.1
%next0.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 1

%s1.1 = load [1 x i8], [1 x i8]* @str.1.1
%sp1.1 = bitcast i8* %next0.1 to [1 x i8]*
store [1 x i8] %s1.1, [1 x i8]* %sp1.1
%next1.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 2

%s2.1 = load [1 x i8], [1 x i8]* @str.2.1
%sp2.1 = bitcast i8* %next1.1 to [1 x i8]*
store [1 x i8] %s2.1, [1 x i8]* %sp2.1
%next2.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 3

%s3.1 = load [1 x i8], [1 x i8]* @str.3.1
%sp3.1 = bitcast i8* %next2.1 to [1 x i8]*
store [1 x i8] %s3.1, [1 x i8]* %sp3.1
%next3.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 4

%s4.1 = load [1 x i8], [1 x i8]* @str.4.1
%sp4.1 = bitcast i8* %next3.1 to [1 x i8]*
store [1 x i8] %s4.1, [1 x i8]* %sp4.1
%next4.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 5

%s5.1 = load [1 x i8], [1 x i8]* @str.5.1
%sp5.1 = bitcast i8* %next4.1 to [1 x i8]*
store [1 x i8] %s5.1, [1 x i8]* %sp5.1
%next5.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 6

%s6.1 = load [1 x i8], [1 x i8]* @str.6.1
%sp6.1 = bitcast i8* %next5.1 to [1 x i8]*
store [1 x i8] %s6.1, [1 x i8]* %sp6.1
%next6.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 7

%s7.1 = load [1 x i8], [1 x i8]* @str.7.1
%sp7.1 = bitcast i8* %next6.1 to [1 x i8]*
```

```
40 store [1 x i8] %s7.1, [1 x i8]* %sp7.1
41 %next7.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 8
42
43 %s8.1 = load [1 x i8], [1 x i8]* @str.8.1
44 %sp8.1 = bitcast i8* %next7.1 to [1 x i8]*
45 store [1 x i8] %s8.1, [1 x i8]* %sp8.1
46 %next8.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 9
47
48 %s9.1 = load [1 x i8], [1 x i8]* @str.9.1
49 %sp9.1 = bitcast i8* %next8.1 to [1 x i8]*
50 store [1 x i8] %s9.1, [1 x i8]* %sp9.1
51 %next9.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 10
52
53 %s10.1 = load [1 x i8], [1 x i8]* @str.10.1
54 %sp10.1 = bitcast i8* %next9.1 to [1 x i8]*
55 store [1 x i8] %s10.1, [1 x i8]* %sp10.1
56 %next10.1 = getelementptr [13 x i8], [13 x i8]* %sp1, i32 0, i32 11
57
58 %s11.1 = load [2 x i8], [2 x i8]* @str.11.1
59 %sp11.1 = bitcast i8* %next10.1 to [2 x i8]*
60 store [2 x i8] %s11.1, [2 x i8]* %sp11.1
61
62 %spi1 = ptrtoint [13 x i8]* %sp1 to i32
63 store i32 %spi1, i32* %6, align 16
```

Code A.2.1: Generated LLVM code loading a shuffled string

A.3 If condition adders

A.3.1 Basic If adder

```
1 ; ModuleID = 'optimized.bc'
2 source_filename = "llvm-link"
```

```
3  target datalayout = "e-m:e-p:32:32-p270:32:32-p271:32:32-p272:64:64-f6
   ↪  4:32:64-f80:32-n8:16:32-S128"
4  target triple = "i386-pc-linux-gnu"
5
6  @fpstt = internal unnamed_addr global i32 0
7  @stack = internal global [4194304 x i32] zeroinitializer, align 16
8  @onUnfallback = common local_unnamed_addr global i1 false
9  @.str1 = private unnamed_addr constant [8 x i8] c"Premier\00"
10 @.str3 = private unnamed_addr constant [9 x i8] c"Deuxieme\00"
11 @.str5 = private unnamed_addr constant [11 x i8] c"Troisième\00"
12
13 ; Function Attrs: mustprogress nofree norecurse nosync nounwind
   ↪  uwtable willreturn writeonly
14 define internal fastcc void @helper_fninit() unnamed_addr #0 {
15   store i32 0, i32* @fpstt, align 16
16   ret void
17 }
18
19 ; Function Attrs: norecurse nounwind uwtable
20 define dso_local noundef i32 @main(i32 noundef %0, i8** noundef %1,
   ↪  i8** noundef %2) local_unnamed_addr #1 {
21   %4 = tail call i32 asm "pushf\0A\09popl $0",
   ↪  "=r,~{dirflag},~{fpsr},~{flags}"() #5
22   tail call fastcc void @helper_fninit() #6
23   %5 = ptrtoint i8** %2 to i32
24   store i32 %5, i32* getelementptr inbounds ([4194304 x i32], [4194304
   ↪  x i32]* @stack, i32 0, i32 4194303), align 4, !tbaa !6
25   %6 = ptrtoint i8** %1 to i32
26   store i32 %6, i32* getelementptr inbounds ([4194304 x i32], [4194304
   ↪  x i32]* @stack, i32 0, i32 4194302), align 8, !tbaa !6
27   store i32 %0, i32* getelementptr inbounds ([4194304 x i32], [4194304
   ↪  x i32]* @stack, i32 0, i32 4194301), align 4, !tbaa !6
28   %7 = tail call i8* @llvm.returnaddress(i32 0)
29   %8 = ptrtoint i8* %7 to i32
30   store i32 %8, i32* getelementptr inbounds ([4194304 x i32], [4194304
   ↪  x i32]* @stack, i32 0, i32 4194300), align 16, !tbaa !6
```

```
31    tail call fastcc void @Func_main(i32 ptrtoint (i32* getelementptr
      ↪  inbounds ([4194304 x i32], [4194304 x i32]* @stack, i32 0, i32
      ↪  4194300) to i32)) #7
32    ret i32 0
33  }
34
35  ; Function Attrs: nofree nosync nounwind readnone willreturn
36  declare i8* @llvm.returnaddress(i32 immarg) #2
37
38  declare dso_local i32 @puts(i8* noundef) local_unnamed_addr #3
39
40  ; Function Attrs: norecurse
41  define internal fastcc void @Func_main(i32 %arg_esp) unnamed_addr #4
      ↪  !retregs !10 {
42    %tmp2_v.i2.i = add i32 %arg_esp, 4
43    %tmp0_v.i3.i = and i32 %arg_esp, -16
44    %1 = inttoptr i32 %arg_esp to i32*
45    %2 = load i32, i32* %1, align 4
46  ;-------------------------------
47  ;--------Basic Condition--------
48    %.not4.i.i = icmp eq i32 0 , 0 ;always true
49    br i1 %.not4.i.i, label %BB_4, label %next4
50  next4:
51  ;-------------------------------
52    %tmp2_v3.i4.i = add i32 %tmp0_v.i3.i, -4
53    %3 = inttoptr i32 %tmp2_v3.i4.i to i32*
54    store i32 %2, i32* %3, align 4
55    %tmp2_v4.i.i = add i32 %tmp0_v.i3.i, -8
56    %4 = inttoptr i32 %tmp2_v4.i.i to i32*
57    store i32 0, i32* %4, align 8
58    %tmp2_v5.i.i = add i32 %tmp0_v.i3.i, -12
59    %5 = inttoptr i32 %tmp2_v5.i.i to i32*
60  ;-------------------------------
61  ;--------Basic Condition--------
62    %.not2.i.i = icmp eq i32 0 , 0 ;always true
```

```
63    br i1 %.not2.i.i, label %BB_2, label %next2
64  next2:
65  ;-------------------------------
66    store i32 0, i32* %5, align 4
67    %tmp2_v6.i.i = add i32 %tmp0_v.i3.i, -16
68    %6 = inttoptr i32 %tmp2_v6.i.i to i32*
69    store i32 %tmp2_v.i2.i, i32* %6, align 16
70  ;-------------------------------
71  ;--------Basic Condition--------
72    %.not6.i.i = icmp eq i32 0 , 0 ;always true
73    br i1 %.not6.i.i, label %BB_6, label %next6
74  next6:
75  ;-------------------------------
76    %tmp2_v7.i.i = add i32 %tmp0_v.i3.i, -20
77    %7 = inttoptr i32 %tmp2_v7.i.i to i32*
78    store i32 134517166, i32* %7, align 4
79    %tmp2_v2.i.i = add i32 %tmp0_v.i3.i, -32
80    %8 = inttoptr i32 %tmp2_v2.i.i to i32*
81    store i32 134520840, i32* %8, align 16
82    %tmp2_v3.i.i = add i32 %tmp0_v.i3.i, -36
83    %9 = inttoptr i32 %tmp2_v3.i.i to i32*
84    store i32 134517188, i32* %9, align 4
85    %arg.i.i = load i32, i32* %8, align 16
86    %10 = inttoptr i32 %arg.i.i to i8*
87    %11 = tail call i32 @puts(i8* nonnull dereferenceable(1) %10)
88    ret void
89  BB_2:
90    %cast2= getelementptr [8 x i8], [8 x i8]* @.str1, i64 0, i64 0
91    call i32 @puts(i8* %cast2)
92    br label %next2
93  BB_4:
94    %cast4= getelementptr [9 x i8], [9 x i8]* @.str3, i64 0, i64 0
95    call i32 @puts(i8* %cast4)
96    br label %next4
97  BB_6:
```

```
98    %cast6= getelementptr [11 x i8], [11 x i8]* @.str5, i64 0, i64 0
99    call i32 @puts(i8* %cast6)
100   br label %next6
101 }
102
103 attributes #0 = { mustprogress nofree norecurse nosync nounwind
      ↪  uwtable willreturn writeonly "frame-pointer"="all"
      ↪  "min-legal-vector-width"="0" "no-trapping-math"="true"
      ↪  "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
      ↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" }
104 attributes #1 = { norecurse nounwind uwtable "frame-pointer"="none"
      ↪  "no-builtins" "no-trapping-math"="true"
      ↪  "stack-protector-buffer-size"="8" "target-cpu"="pentium4"
      ↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
      ↪  "tune-cpu"="generic" }
105 attributes #2 = { nofree nosync nounwind readnone willreturn }
106 attributes #3 = { "frame-pointer"="none" "no-builtins"
      ↪  "no-trapping-math"="true" "stack-protector-buffer-size"="8"
      ↪  "target-cpu"="pentium4"
      ↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
      ↪  "tune-cpu"="generic" }
107 attributes #4 = { norecurse }
108 attributes #5 = { nounwind readnone }
109 attributes #6 = { nobuiltin nounwind "no-builtins" }
110 attributes #7 = { nounwind }
111
112 !llvm.ident = !{!0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0,
      ↪  !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0,
      ↪  !0, !0, !0, !0}
113 !llvm.module.flags = !{!1, !2, !3, !4, !5}
114
115 !0 = !{!"clang version 14.0.0"}
116 !1 = !{i32 1, !"wchar_size", i32 4}
117 !2 = !{i32 7, !"PIC Level", i32 2}
118 !3 = !{i32 7, !"uwtable", i32 1}
```

```
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{i32 1, !"NumRegisterParameters", i32 0}
!6 = !{!7, !7, i64 0}
!7 = !{!"int", !8, i64 0}
!8 = !{!"omnipotent char", !9, i64 0}
!9 = !{!"Simple C++ TBAA"}
!10 = !{i32 0, i32 0, i32 0}
```

Code A.3.1: LLVM code after lifting a "Hello World" program, and adding three basic if conditions.

A.3.2 Random if adder

```
; ModuleID = 'optimized.bc'
source_filename = "llvm-link"
target datalayout = "e-m:e-p:32:32-p270:32:32-p271:32:32-p272:64:64-f64:32:64-f80:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

@fpstt = internal unnamed_addr global i32 0
@stack = internal global [4194304 x i32] zeroinitializer, align 16
@onUnfallback = common local_unnamed_addr global i1 false

; Function Attrs: mustprogress nofree norecurse nosync nounwind uwtable willreturn writeonly
define internal fastcc void @helper_fninit() unnamed_addr #0 {
  store i32 0, i32* @fpstt, align 16
  ret void
}

; Function Attrs: norecurse nounwind uwtable
define dso_local noundef i32 @main(i32 noundef %0, i8** noundef %1, i8** noundef %2) local_unnamed_addr #1 {
  %4 = tail call i32 asm "pushf\0A\09popl $0", "=r,~{dirflag},~{fpsr},~{flags}"() #5
```

```
19    tail call fastcc void @helper_fninit() #6
20    %5 = ptrtoint i8** %2 to i32
21    store i32 %5, i32* getelementptr inbounds ([4194304 x i32], [4194304
      ↪  x i32]* @stack, i32 0, i32 4194303), align 4, !tbaa !6
22    %6 = ptrtoint i8** %1 to i32
23    store i32 %6, i32* getelementptr inbounds ([4194304 x i32], [4194304
      ↪  x i32]* @stack, i32 0, i32 4194302), align 8, !tbaa !6
24    store i32 %0, i32* getelementptr inbounds ([4194304 x i32], [4194304
      ↪  x i32]* @stack, i32 0, i32 4194301), align 4, !tbaa !6
25    %7 = tail call i8* @llvm.returnaddress(i32 0)
26    %8 = ptrtoint i8* %7 to i32
27    store i32 %8, i32* getelementptr inbounds ([4194304 x i32], [4194304
      ↪  x i32]* @stack, i32 0, i32 4194300), align 16, !tbaa !6
28    tail call fastcc void @Func_main(i32 ptrtoint (i32* getelementptr
      ↪  inbounds ([4194304 x i32], [4194304 x i32]* @stack, i32 0, i32
      ↪  4194300) to i32)) #7
29    ret i32 0
30  }
31
32  ; Function Attrs: nofree nosync nounwind readnone willreturn
33  declare i8* @llvm.returnaddress(i32 immarg) #2
34
35  declare dso_local i32 @puts(i8* noundef) local_unnamed_addr #3
36
37  ;-------------------------------
38  ;--------Extra functions--------
39  declare i32 @rand() local_unnamed_addr  noinline
40
41  declare void @srand(i32) local_unnamed_addr  noinline
42
43  declare i32 @time(i32*) local_unnamed_addr noinline
44
45  ;-------------------------------
46  ; Function Attrs: norecurse
47  define internal fastcc void @Func_main(i32 %arg_esp) unnamed_addr #4
    ↪  !retregs !10 {
```

```
48    %tmp2_v.i6.i = add i32 %arg_esp, 4
49    %time1 = tail call i32 @time(i32* null)
50    tail call void @srand(i32 %time1)
51    %rand_init6 = tail call i32 @rand()
52    %rand_fin7 = srem i32 %rand_init6, 5
53    %rand_init2 = tail call i32 @rand()
54    %rand_fin3 = srem i32 %rand_init2, 5
55    %tmp0_v.i7.i = and i32 %arg_esp, -16
56    %1 = inttoptr i32 %arg_esp to i32*
57    %2 = load i32, i32* %1, align 4
58    %tmp2_v3.i9.i = add i32 %tmp0_v.i7.i, -4
59    %3 = inttoptr i32 %tmp2_v3.i9.i to i32*
60    store i32 %2, i32* %3, align 4
61    %tmp2_v4.i.i = add i32 %tmp0_v.i7.i, -8
62  ;-------------------------------
63  ;----------Random Cond----------
64    %.not4.i.i = icmp eq i32 4 , %rand_fin3
65    br i1 %.not4.i.i, label %next4, label %BB_4
66  next4:
67  ;-------------------------------
68    %4 = inttoptr i32 %tmp2_v4.i.i to i32*
69    store i32 0, i32* %4, align 8
70    %tmp2_v5.i11.i = add i32 %tmp0_v.i7.i, -12
71    %5 = inttoptr i32 %tmp2_v5.i11.i to i32*
72    store i32 0, i32* %5, align 4
73    %tmp2_v6.i.i = add i32 %tmp0_v.i7.i, -16
74    %6 = inttoptr i32 %tmp2_v6.i.i to i32*
75    store i32 %tmp2_v.i6.i, i32* %6, align 16
76    %tmp2_v7.i.i = add i32 %tmp0_v.i7.i, -20
77    %7 = inttoptr i32 %tmp2_v7.i.i to i32*
78    store i32 134517166, i32* %7, align 4
79    %tmp2_v2.i.i = add i32 %tmp0_v.i7.i, -32
80    %8 = inttoptr i32 %tmp2_v2.i.i to i32*
81    store i32 134520840, i32* %8, align 16
82  ;-------------------------------
```

```
;----------Random Cond----------
  %.not8.i.i = icmp eq i32 3 , %rand_fin7
  br i1 %.not8.i.i, label %next8, label %BB_4
next8:
;-------------------------------
  %tmp2_v3.i.i = add i32 %tmp0_v.i7.i, -36
  %9 = inttoptr i32 %tmp2_v3.i.i to i32*
  store i32 134517188, i32* %9, align 4
  %arg.i.i = load i32, i32* %8, align 16
  %10 = inttoptr i32 %arg.i.i to i8*
  %11 = tail call i32 @puts(i8* nonnull dereferenceable(1) %10)
  ret void
BB_4:
  ret void
}

attributes #0 = { mustprogress nofree norecurse nosync nounwind
↪  uwtable willreturn writeonly "frame-pointer"="all"
↪  "min-legal-vector-width"="0" "no-trapping-math"="true"
↪  "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" }
attributes #1 = { norecurse nounwind uwtable "frame-pointer"="none"
↪  "no-builtins" "no-trapping-math"="true"
↪  "stack-protector-buffer-size"="8" "target-cpu"="pentium4"
↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
↪  "tune-cpu"="generic" }
attributes #2 = { nofree nosync nounwind readnone willreturn }
attributes #3 = { "frame-pointer"="none" "no-builtins"
↪  "no-trapping-math"="true" "stack-protector-buffer-size"="8"
↪  "target-cpu"="pentium4"
↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
↪  "tune-cpu"="generic" }
attributes #4 = { norecurse }
attributes #5 = { nounwind readnone }
attributes #6 = { nobuiltin nounwind "no-builtins" }
```

```
attributes #7 = { nounwind }

!llvm.ident = !{!0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0,
↪  !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0,
↪  !0, !0, !0, !0}
!llvm.module.flags = !{!1, !2, !3, !4, !5}

!0 = !{!"clang version 14.0.0"}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{i32 1, !"NumRegisterParameters", i32 0}
!6 = !{!7, !7, i64 0}
!7 = !{!"int", !8, i64 0}
!8 = !{!"omnipotent char", !9, i64 0}
!9 = !{!"Simple C++ TBAA"}
!10 = !{i32 0, i32 0, i32 0}


```

Code A.3.2: LLVM code after lifting a "Hello World" program, and adding two random if conditions, with the random mod 5.

A.4 System calls replacement

```
;; Mutation 2
; ModuleID = 'optimized.bc'
source_filename = "llvm-link"
target datalayout = "e-m:e-p:32:32-p270:32:32-p271:32:32-p272:64:64-f6
↪  4:32:64-f80:32-n8:16:32-S128"
target triple = "i386-pc-linux-gnu"

```

```
@fpstt = internal unnamed_addr global i32 0
@stack = internal global [4194304 x i32] zeroinitializer, align 16
@onUnfallback = common local_unnamed_addr global i1 false
@.str1 = private unnamed_addr constant [1 x i8] c"\0a"

; Function Attrs: mustprogress nofree norecurse nosync nounwind
↪  uwtable willreturn writeonly
define internal fastcc void @helper_fninit() unnamed_addr #0 {
  store i32 0, i32* @fpstt, align 16
  ret void
}

; Function Attrs: norecurse nounwind uwtable
define dso_local noundef i32 @main(i32 noundef %0, i8** noundef %1,
↪  i8** noundef %2) local_unnamed_addr #1 {
  %4 = tail call i32 asm "pushf\0A\09popl $0",
↪  "=r,~{dirflag},~{fpsr},~{flags}"() #5
  tail call fastcc void @helper_fninit() #6
  %5 = ptrtoint i8** %2 to i32
  store i32 %5, i32* getelementptr inbounds ([4194304 x i32], [4194304
↪  x i32]* @stack, i32 0, i32 4194303), align 4, !tbaa !6
  %6 = ptrtoint i8** %1 to i32
  store i32 %6, i32* getelementptr inbounds ([4194304 x i32], [4194304
↪  x i32]* @stack, i32 0, i32 4194302), align 8, !tbaa !6
  store i32 %0, i32* getelementptr inbounds ([4194304 x i32], [4194304
↪  x i32]* @stack, i32 0, i32 4194301), align 4, !tbaa !6
  %7 = tail call i8* @llvm.returnaddress(i32 0)
  %8 = ptrtoint i8* %7 to i32
  store i32 %8, i32* getelementptr inbounds ([4194304 x i32], [4194304
↪  x i32]* @stack, i32 0, i32 4194300), align 16, !tbaa !6
  tail call fastcc void @Func_main(i32 ptrtoint (i32* getelementptr
↪  inbounds ([4194304 x i32], [4194304 x i32]* @stack, i32 0, i32
↪  4194300) to i32)) #7
  ret i32 0
}
```

```
33
34  ; Function Attrs: nofree nosync nounwind readnone willreturn
35  declare i8* @llvm.returnaddress(i32 immarg) #2
36
37  declare dso_local i32 @puts(i8* noundef) local_unnamed_addr #3
38
39  ; Function Attrs: naked noinline
40  declare dso_local i32 @printf(i8* noundef) local_unnamed_addr  naked
    ↪  noinline "frame-pointer"="none" "no-builtins"
    ↪  "no-trapping-math"="true" "stack-protector-buffer-size"="8"
    ↪  "target-cpu"="pentium4"
    ↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
    ↪  "tune-cpu"="generic"
41  ; Function Attrs: norecurse
42  define internal fastcc void @Func_main(i32 %arg_esp) unnamed_addr #4
    ↪  !retregs !10 {
43    %tmp2_v.i.i = add i32 %arg_esp, 4
44    %tmp0_v.i.i = and i32 %arg_esp, -16
45    %1 = inttoptr i32 %arg_esp to i32*
46    %2 = load i32, i32* %1, align 4
47    %tmp2_v3.i.i = add i32 %tmp0_v.i.i, -4
48    %3 = inttoptr i32 %tmp2_v3.i.i to i32*
49    store i32 %2, i32* %3, align 4
50    %tmp2_v4.i.i = add i32 %tmp0_v.i.i, -8
51    %4 = inttoptr i32 %tmp2_v4.i.i to i32*
52    store i32 0, i32* %4, align 8
53    %tmp2_v5.i.i = add i32 %tmp0_v.i.i, -12
54    %5 = inttoptr i32 %tmp2_v5.i.i to i32*
55    store i32 0, i32* %5, align 4
56    %tmp2_v6.i.i = add i32 %tmp0_v.i.i, -16
57    %6 = inttoptr i32 %tmp2_v6.i.i to i32*
58    store i32 %tmp2_v.i.i, i32* %6, align 16
59    %tmp2_v7.i.i = add i32 %tmp0_v.i.i, -20
60    %7 = inttoptr i32 %tmp2_v7.i.i to i32*
61    store i32 134517166, i32* %7, align 4
```

```
62    %tmp2_v2.i.i = add i32 %tmp0_v.i.i, -32
63    %8 = inttoptr i32 %tmp2_v2.i.i to i32*
64    store i32 134520840, i32* %8, align 16
65    %tmp2_v3.i11.i = add i32 %tmp0_v.i.i, -36
66    %9 = inttoptr i32 %tmp2_v3.i11.i to i32*
67    store i32 134517188, i32* %9, align 4
68    %arg.i.i = load i32, i32* %8, align 16
69    %10 = inttoptr i32 %arg.i.i to i8*
70  ;-------------------------------
71  ;---Replaced puts with printf---
72    %fp2 = tail call i32 @printf(i8* nonnull dereferenceable(1) %10)
    ↪  nobuiltin nounwind "no-builtins" , !funcname !11
73    %cast2= getelementptr [1 x i8], [1 x i8]* @.str1, i64 0, i64 0
74    %11 = tail call i32 @printf(i8* nonnull dereferenceable(1) %cast2)
    ↪  nobuiltin nounwind "no-builtins" , !funcname !11
75  ;-------------------------------
76    ret void
77  }
78
79  attributes #0 = { mustprogress nofree norecurse nosync nounwind
    ↪  uwtable willreturn writeonly "frame-pointer"="all"
    ↪  "min-legal-vector-width"="0" "no-trapping-math"="true"
    ↪  "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
    ↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" }
80  attributes #1 = { norecurse nounwind uwtable "frame-pointer"="none"
    ↪  "no-builtins" "no-trapping-math"="true"
    ↪  "stack-protector-buffer-size"="8" "target-cpu"="pentium4"
    ↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
    ↪  "tune-cpu"="generic" }
81  attributes #2 = { nofree nosync nounwind readnone willreturn }
82  attributes #3 = { "frame-pointer"="none" "no-builtins"
    ↪  "no-trapping-math"="true" "stack-protector-buffer-size"="8"
    ↪  "target-cpu"="pentium4"
    ↪  "target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87"
    ↪  "tune-cpu"="generic" }
```

```
attributes #4 = { norecurse }
attributes #5 = { nounwind readnone }
attributes #6 = { nobuiltin nounwind "no-builtins" }
attributes #7 = { nounwind }

!llvm.ident = !{!0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0,
↪  !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0, !0,
↪  !0, !0, !0, !0}
!llvm.module.flags = !{!1, !2, !3, !4, !5}

!0 = !{!"clang version 14.0.0"}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{i32 7, !"uwtable", i32 1}
!4 = !{i32 7, !"frame-pointer", i32 2}
!5 = !{i32 1, !"NumRegisterParameters", i32 0}
!6 = !{!7, !7, i64 0}
!7 = !{!"int", !8, i64 0}
!8 = !{!"omnipotent char", !9, i64 0}
!9 = !{!"Simple C++ TBAA"}
!10 = !{i32 0, i32 0, i32 0}
!11 = !{!"printf"}
```

Code A.4.1: LLVM code after lifting a "Hello World" program, and replacing the `puts` function with `printf`

A.5 Base64 decoding function

```
@base64_decode_table = internal constant [64 x i8]
↪  c"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"

define internal fastcc i8* @base64_decode(i8* %cipher.ptr, i32 %len) {
  %counts.ptr = alloca i8
```

```
  store i8 0, i8* %counts.ptr
  %plain_len.tmp = mul i32 %len, 3
  %plain_len = sdiv i32 %plain_len.tmp, 4
  %buffer = alloca [4 x i8]
  %plain = alloca i8, i32 %plain_len
  %i.ptr = alloca i32
  %p.ptr = alloca i32
  store i32 0, i32* %i.ptr
  store i32 0, i32* %p.ptr
  br label %loop.letters.top
  loop.letters.top:
    %i = load i32, i32* %i.ptr
    %cmp = icmp ult i32 %i, %len
    br i1 %cmp, label %loop.letters.body, label %loop.letters.end
  loop.letters.body:
    %k.ptr = alloca i8
    store i8 0, i8* %k.ptr
    br label %loop.decode.top
    loop.decode.top:
      %k = load i8, i8* %k.ptr
      %cmp1 = icmp ult i8 %k, 64
      br i1 %cmp1, label %loop.decode.body, label %loop.decode.end
      %map_k.ptr = getelementptr inbounds [64 x i8], [64 x i8]*
↪   @base64_decode_table, i32 0, i8 %k
      %map_k = load i8, i8* %map_k.ptr, align 1
      %cipher_i = getelementptr inbounds i8, i8* %cipher.ptr, i32 %i
      %cipher = load i8, i8* %cipher_i, align 1
      %cmp2 = icmp ne i8 %map_k, %cipher
      %cmp3 = and i1 %cmp1, %cmp2
      br i1 %cmp3, label %loop.decode.body, label %loop.decode.end
    loop.decode.body:
      %k.inc = add i8 %k, 1
      store i8 %k.inc, i8* %k.ptr
      br label %loop.decode.top
    loop.decode.end:
```

```
39      %counts = load i8, i8* %counts.ptr
40      %buffer_counts = getelementptr inbounds [4 x i8], [4 x i8]*
    ↪  %buffer, i32 0, i8 %counts
41      store i8 %k, i8* %buffer_counts
42      %counts.inc = add i8 %counts, 1
43      store i8 %counts.inc, i8* %counts.ptr
44      %counts_is_4 = icmp eq i8 %counts.inc, 4
45      br i1 %counts_is_4, label %if.counts_4.true, label
    ↪  %if.counts_4.false
46      if.counts_4.true:
47        %buffer_0.ptr = getelementptr inbounds [4 x i8], [4 x i8]*
    ↪  %buffer, i32 0, i8 0
48        %buffer_1.ptr = getelementptr inbounds [4 x i8], [4 x i8]*
    ↪  %buffer, i32 0, i8 1
49        %buffer_0 = load i8, i8* %buffer_0.ptr
50        %buffer_1 = load i8, i8* %buffer_1.ptr
51        %buffer_0.shifted = shl i8 %buffer_0, 2
52        %buffer_1.shifted = lshr i8 %buffer_1, 4
53        %buffer.add = add i8 %buffer_0.shifted, %buffer_1.shifted
54        %p = load i32, i32* %p.ptr
55        %plain_p = getelementptr inbounds i8, i8* %plain, i32 %p
56        store i8 %buffer.add, i8* %plain_p
57        %p.inc = add i32 %p, 1
58        store i32 %p.inc, i32* %p.ptr
59        %buffer_2.ptr = getelementptr inbounds [4 x i8], [4 x i8]*
    ↪  %buffer, i32 0, i8 2
60        %buffer_2 = load i8, i8* %buffer_2.ptr
61        %buffer_2_is_not_64 = icmp ne i8 %buffer_2, 64
62        br i1 %buffer_2_is_not_64, label %if.buffer_2_is_not_64.true,
    ↪  label %if.buffer_2_is_not_64.false
63        if.buffer_2_is_not_64.true:
64          %buffer_1.shifted_1 = shl i8 %buffer_1, 4
65          %buffer_2.shifted = lshr i8 %buffer_2, 2
66          %buffer_1_2.add = add i8 %buffer_1.shifted_1, %buffer_2.shifted
67          %p_1 = load i32, i32* %p.ptr
```

```
68          %plain_p_1 = getelementptr inbounds i8, i8* %plain, i32 %p_1
69          store i8 %buffer_1_2.add, i8* %plain_p_1
70          %p.inc_1 = add i32 %p_1, 1
71          store i32 %p.inc_1, i32* %p.ptr
72          br label %if.buffer_2_is_not_64.false
73        if.buffer_2_is_not_64.false:
74        %buffer_3.ptr = getelementptr inbounds [4 x i8], [4 x i8]*
    ↪  %buffer, i32 0, i8 3
75        %buffer_3 = load i8, i8* %buffer_3.ptr
76        %buffer_3_is_not_64 = icmp ne i8 %buffer_3, 64
77        br i1 %buffer_3_is_not_64, label %if.buffer_3_is_not_64.true,
    ↪  label %if.buffer_3_is_not_64.false
78        if.buffer_3_is_not_64.true:
79          %buffer_2.shifted_1 = shl i8 %buffer_2, 6
80          %buffer_2_3.add = add i8 %buffer_2.shifted_1, %buffer_3
81          %p_2 = load i32, i32* %p.ptr
82          %plain_p_2 = getelementptr inbounds i8, i8* %plain, i32 %p_2
83          store i8 %buffer_2_3.add, i8* %plain_p_2
84          %p.inc_2 = add i32 %p_2, 1
85          store i32 %p.inc_2, i32* %p.ptr
86          br label %if.buffer_3_is_not_64.false
87        if.buffer_3_is_not_64.false:
88        store i8 0, i8* %counts.ptr
89        br label %if.counts_4.false
90      if.counts_4.false:
91      %i.inc = add i32 %i, 1
92      store i32 %i.inc, i32* %i.ptr
93      br label %loop.letters.top
94    loop.letters.end:
95    %p_3 = load i32, i32* %p.ptr
96    %plain_p_3 = getelementptr inbounds i8, i8* %plain, i32 %p_3
97    store i8 0, i8* %plain_p_3
98    ret i8* %plain
99  }
```

Code A.5.1: LLVM function decoding a base64 cipher

A.6 VM detection code

```
1
2  int detect_vm() {
3
4        /*run number_of_cores() function first,
5      as it runs on both windows and unix machines.
6      */
7      number_of_cores();
8
9
10     //run the dmesg command and pipe to find hypervisor, 34 is how
   ↪   long string should be.
11     run_command("sudo dmesg | grep -i hypervisor", "[   0.000000
   ↪   Hypervisor detected]", 34);
12
13     //run dmidecode command and find system manufacturer, 6 is how
   ↪   long the string should be.
14     run_command("sudo dmidecode -s system-manufacturer", "VMware", 6);
15
16     /*If vm_score is less than 2, we are likely running on physical
   ↪   hardware*/
17     if(vm_score < 2){
18       return 0;
19     } else {
20       return 1;
21     }
22
23 }
24
25 /* number_of_cores serves the purpose of checking to see how many
26    cores the system is running on. If the core count is less than
```

```
27      one, there is a very good chance we are running on a virtual
28      machine. */
29
30  void number_of_cores() {
31      //run sysconf function outlined in the man pages.
32      if(sysconf(_SC_NPROCESSORS_ONLN) <= 1){
33          //check if number of processors is less than or equal to one.
    ↪  If it is,
34          //we assume virtual, and increment vm_score by 1.
35          vm_score++;
36      }
37  } //end of number_of_cores()
38
39
40  /* run_command serves the purposes of running terminal commands
41      within both linux and windows environments.
42      We use this for dmesg, dmidecode, and systeminfo.
43
44      run_command() takes three parameters: cmd, detphrase, and dp_length.
45      cmd is the command to run such as systeminfo or dmesg or dmidecode
46
47      detphrase is the string to check for within the command. If we run
48      dmidecode and we're looking for "VMWARE", the detphrase is "VMWARE".
49
50      dp_length is the length of the string we want to specify. This
    ↪  allows
51      us to take substrings of the system output in order to have cleaner,
52      more readable code.
53
54      The benefits of using this run_command() function over a system()
    ↪  call
55      is that it allows us to save the output of system calls as well as
    ↪  take
56      substrings.
57  */
```

```
58 void run_command(char *cmd, char *detphrase, int dp_length){
59     #define BUFSIZE 128
60     char buf[BUFSIZE];
61     FILE *fp;
62 
63     /*popen() is essentially the same as system()
64     but it saves the output to a file. If the output
65     is null, the command didn't work.
66     */
67     if((fp = popen(cmd, "r")) == NULL){
68         printf("Error");
69     }
70 
71 
72     if(fgets(buf, BUFSIZE, fp) != NULL){
73         char detection[(dp_length +1 )]; //one extra char for null
    ↪  terminator
74         strncpy(detection, detphrase, dp_length);
75         detection[dp_length] = '\0'; //place the null terminator
76 
77         if(strcmp(detphrase, detection) == 0){ //0 means detphrase =
    ↪  detection
78             vm_score++; //increment the vm_score variable.
79         }
80     }
81 
82     if(pclose(fp)){
83         printf("Command not found or exited with error status \n");
84     }
85 }
```

Code A.6.1: Adapted code from JakenHerman [18]

Appendix B

Toolchain code

B.1 ArgumentParserTC

```
1
2  import argparse
3  from enum import Enum
4
5  try:
6      from SemaSCDG.helper.ArgumentParserSCDG import ArgumentParserSCDG
7      from SemaClassifier.helper.ArgumentParserClassifier import
       ↪  ArgumentParserClassifier
8      from src.SemaMutator.helper.ArgumentParserMutator import
       ↪  ArgumentParserMutator
9  except:
10     from src.SemaSCDG.helper.ArgumentParserSCDG import
       ↪  ArgumentParserSCDG
11     from src.SemaClassifier.helper.ArgumentParserClassifier import
       ↪  ArgumentParserClassifier
12     from src.SemaMutator.helper.ArgumentParserMutator import
       ↪  ArgumentParserMutator
13
14 class ArgumentParserTC:
15
16     def __init__(self,tcw,tcc,tcm):
```

```
17          self.tool_mutate = tcm
18          self.args_parser_mutator = ArgumentParserMutator(tcm)
19          self.tool_scdg = tcw
20          self.args_parser_scdg = ArgumentParserSCDG(tcw)
21          self.tool_classifier = tcc
22          self.args_parser_class = ArgumentParserClassifier(tcc)
23          # self.parser =
            ↪  argparse.ArgumentParser(conflict_handler='resolve',
24          #
            ↪  parents=[self.args_parser_mutator.parser,
25          #
            ↪  self.args_parser_scdg.parser,
26          #
            ↪  self.args_parser_class.parser])
27          self.tools = Enum('tools', ['MUTATOR', 'SCDG', 'CLASSIFIER'])
28          self._enabled = []
29
30      def reset(self):
31          self._enabled = []
32
33      def enable(self, tool):
34          if tool not in self.tools:
35              raise  ValueError(f"{tool} is not a valid tool type.")
36          if tool not in self._enabled:
37              self._enabled.append(tool)
38
39      def disable(self, tool):
40          if tool not in self.tools:
41              raise  ValueError(f"{tool} is not a valid tool type.")
42          if tool in self._enabled:
43              self._enabled.remove(tool)
44
45      # TODO conflict with --exp_dir and binaries arguments
46      # def parse_arguments(self, allow_unk=False,args_list=None):
47      #     args = None
```

```
48      #     if not allow_unk:
49      #         args = self.parser.parse_args(args_list)
50      #     else:
51      #         args, unknown = self.parser.parse_known_args(args_list)
52      #     return args
53
54      def parse_arguments(self, allow_unk=False,args_list=None):
55          args = {}
56          for tool in self._enabled:
57              if tool == self.tools.MUTATOR:
58                  args[tool] = self.parse_arguments_with(self.args_parse
                        ↪ r_mutator.parser, allow_unk,
                        ↪ args_list)
59              elif tool == self.tools.SCDG:
60                  args[tool] = self.parse_arguments_with(self.args_parse
                        ↪ r_scdg.parser, allow_unk,
                        ↪ args_list)
61              elif tool == self.tools.CLASSIFIER:
62                  args[tool] = self.parse_arguments_with(self.args_parse
                        ↪ r_class.parser, allow_unk,
                        ↪ args_list)
63          return args
64
65      def parse_arguments_with(self, parser,
        ↪ allow_unk=False,args_list=None):
66          args = None
67          if not allow_unk:
68              args = parser.parse_args(args_list)
69          else:
70              args, unknown = parser.parse_known_args(args_list)
71          return args
```

B.2 SEMAServer

```
@app.route('/index.html', methods = ['GET', 'POST'])
def serve_index():
    """
    It creates a folder for the project, and then calls the upload function
    :return: the upload function.
    """
    if request.method == 'POST':
        SemaServer.log.info(request.form)

        MUTATOR = SemaServer.sema.args_parser.tools.MUTATOR
        SCDG = SemaServer.sema.args_parser.tools.SCDG
        CLASSIFIER = SemaServer.sema.args_parser.tools.CLASSIFIER

        exp_args = []
        SemaServer.sema.args_parser.reset()

        # TODO dir per malware
        if "mutator_enable" in request.form:
            SemaServer.sema.args_parser.enable(MUTATOR)
            mut_args, exp_args_mut, exp_args_mut_str = SemaServer.get_mutator_args(request)
            exp_args += exp_args_mut
        if "scdg_enable" in request.form:
            SemaServer.sema.args_parser.enable(SCDG)
            scdg_args, exp_args_scdg, exp_args_scdg_str = SemaServer.get_scdg_args(request)
            exp_args += exp_args_scdg
        if "class_enable" in request.form:
            SemaServer.sema.args_parser.enable(CLASSIFIER)
            class_args, exp_args_class, exp_args_class_str = SemaServer.get_class_args(request)
            exp_args += exp_args_class
```

```
30          if "fl_enable" in request.form: # TODO refactor + implement
31              fl_args, exp_args_fl, exp_args_fl_str =
                ↪  SemaServer.get_fl_args(request)
32              exp_args += exp_args_fl
33
34          SemaServer.log.info(exp_args)
35
36          args = SemaServer.sema.args_parser.parse_arguments(args_list=e
            ↪  xp_args,allow_unk=False) #
            ↪  TODO
37
38          SemaServer.log.info(args)
39
40          ##
41          # Here we link the individual argument about input/output
            ↪  folder so they match
42          # Typically: [mutator] -> [scdg] -> [class]
43          ##
44          if "mutator_enable" in request.form and
            ↪  args[MUTATOR].binaries_mutated == "output/runs/": #FIXME I
            ↪  have no idea What i'm doing
45              if "scdg_enabled" in request.form:
46                  args[SCDG].binary = args[MUTATOR].binaries_mutated
47          elif "scdg_enable" in request.form and args[SCDG].exp_dir ==
            ↪  "output/runs/":
48              SemaServer.sema.current_exp_dir = len(glob.glob("src/" +
                ↪  args[SCDG].exp_dir + "/*")) + 1
49              args[SCDG].exp_dir = "src/" + args[SCDG].exp_dir +
                ↪  str(SemaServer.sema.current_exp_dir) + "/"
50              args[SCDG].dir = "src/" + args[SCDG].dir +
                ↪  str(SemaServer.sema.current_exp_dir) + "/"
51              args[CLASSIFIER].binaries = args[SCDG].exp_dir
52          elif "class_enable" in request.form and
            ↪  args[CLASSIFIER].binaries == "output/runs/":
53              SemaServer.sema.current_exp_dir = len(glob.glob("src/" +
                ↪  args[SCDG].exp_dir + "/*")) + 1
```

```
54              exp_dir = "src/" + args[SCDG].exp_dir +
                ↪  str(SemaServer.sema.current_exp_dir) + "/"
55              args[CLASSIFIER].binaries = exp_dir
56          else:
57              SemaServer.sema.current_exp_dir =
                ↪  int(args[CLASSIFIER].binaries.split("/")[-1]) # TODO
58
59          ##
60          # Here we start the experiments
61          ##
62
63          if "mutator_enable" in request.form:
64              SemaServer.sema.tool_mutator.args = args[MUTATOR]
65              SemaServer.sema.args_parser.args_parser_mutator.update_too⌋
                ↪  l(args[MUTATOR])
66              SemaServer.exps.append(threading.Thread(target=SemaServer.⌋
                ↪  sema.tool_mutator.print_args))
67              SemaServer.exps.append(threading.Thread(target=SemaServer.⌋
                ↪  sema.tool_mutator.start_mutation))
68
69          if "scdg_enable" in request.form:
70              SemaServer.sema.tool_classifier.args = args[SCDG]
71              SemaServer.sema.args_parser.args_parser_scdg.update_tool(a⌋
                ↪  rgs[SCDG])
72              csv_scdg_file = "src/output/runs/"+str(SemaServer.sema.cur⌋
                ↪  rent_exp_dir)+"/" +
                ↪  "scdg.csv"
73              SemaServer.log.info(csv_scdg_file)
74              SemaServer.exps.append(threading.Thread(target=SemaServer.⌋
                ↪  sema.tool_scdg.save_conf,
                ↪  args=([scdg_args,"src/output/runs/"+str(SemaServer.sem⌋
                ↪  a.current_exp_dir)+"/"])))
75              SemaServer.exps.append(threading.Thread(target=SemaServer.⌋
                ↪  sema.tool_scdg.start_scdg, args=([args[SCDG], False,
                ↪  csv_scdg_file])))
```

```
76
77          if "class_enable" in request.form:
78              SemaServer.sema.tool_classifier.args = args[CLASSIFIER]
79              SemaServer.sema.args_parser.args_parser_class.update_tool(
                ↪  args[CLASSIFIER])
80              csv_class_file =  "src/output/runs/"+str(SemaServer.sema.c
                ↪  urrent_exp_dir)+"/" +
                ↪  "classifier.csv"
81              SemaServer.exps.append(threading.Thread(target=SemaServer.
                ↪  sema.tool_classifier.init, args=([args[SCDG].exp_dir,
                ↪  [], csv_class_file]))) # TODO familly
82              SemaServer.exps.append(threading.Thread(target=SemaServer.
                ↪  sema.tool_classifier.save_conf,args=([class_args,"src/
                ↪  output/runs/"+str(SemaServer.sema.current_exp_dir)+"/"
                ↪  ])))
83              SemaServer.exps.append(threading.Thread(target=SemaServer.
                ↪  sema.tool_classifier.train,
                ↪  args=()))
84              if SemaServer.sema.tool_classifier.mode ==
                ↪  "classification":
85                  SemaServer.exps.append(threading.Thread(target=SemaSer
                    ↪  ver.sema.tool_classifier.classify,
                    ↪  args=()))
86              else:
87                  SemaServer.exps.append(threading.Thread(target=SemaSer
                    ↪  ver.sema.tool_classifier.detect,
                    ↪  args=()))
88              SemaServer.exps.append(threading.Thread(target=SemaServer.
                ↪  sema.tool_classifier.save_csv,
                ↪  args=()))
89
90          if "fl_enable" in request.form:
91              pass
92
93          try:
```

```
94              os.mkdir("src/output/runs/"+str(SemaServer.sema.current_ex
                ↪  p_dir)+"/")
95              SemaServer.sema.tool_scdg.current_exp_dir
                ↪  =SemaServer.sema.current_exp_dir
96          except:
97              pass
98          threading.Thread(target=SemaServer.manage_exps,
            ↪  args=([args])).start()
99
100         return render_template('index.html',
101                                actions_mutator=SemaServer.actions_mutator,
102                                actions_scdg=SemaServer.actions_scdg,
103                                actions_classifier=SemaServer.actions_clas
                                   ↪  sifier,
104                                progress=0) # TODO Ortt
105     else:
106         return render_template('index.html',
107                                actions_mutator=SemaServer.actions_mutator,
108                                actions_scdg=SemaServer.actions_scdg,
109                                actions_classifier=SemaServer.actions_clas
                                   ↪  sifier,
110                                progress=0)
```

Appendix C

Evaluation

C.1 Program mutated for evaluation

```
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>

void print_func(){
    char *hello = "I am evil!!!\n";
        char *bye = "Bye Bye\n";
    printf("%s", hello);
    return;
}

void print_file(){
    FILE *fp;
    fp = fopen("text_file3.txt", "w+");
    if( fp == NULL ) {
    fprintf(stderr, "Couldn't open:\n");
    exit(1);
}
```

```
22      char add_text[] = "I altered the file!";
23
24      fwrite(add_text, sizeof(char), sizeof(add_text), fp);
25
26      fclose(fp);
27
28      return;
29  }
30
31  void socket_func(){
32      int listen = 0;
33      listen = socket(AF_INET, SOCK_STREAM, 0);
34
35      close(listen);
36      return;
37  }
38
39  int main(int argc, char **argv){
40      if(!strcmp(argv[1],"1")){
41          print_func();
42      }
43      else if (!strcmp(argv[1],"2"))
44      {
45          print_file();
46      }
47      else if (!strcmp(argv[1],"3"))
48      {
49          socket_func();
50      }
51
52          return 0;
53  }
```

C.1.1 Basic Program for the Proof of work puts-replace

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv){
        puts("Bonjour à tous je suis ");
    if(!strcmp(argv[1],"1")){
        puts("countent !\n");
    } else {
                puts("I am evil !!!\n");
        }
        return 0;
}
```

C.2 Complete SCDG

C.2.1 Cleanware Adder

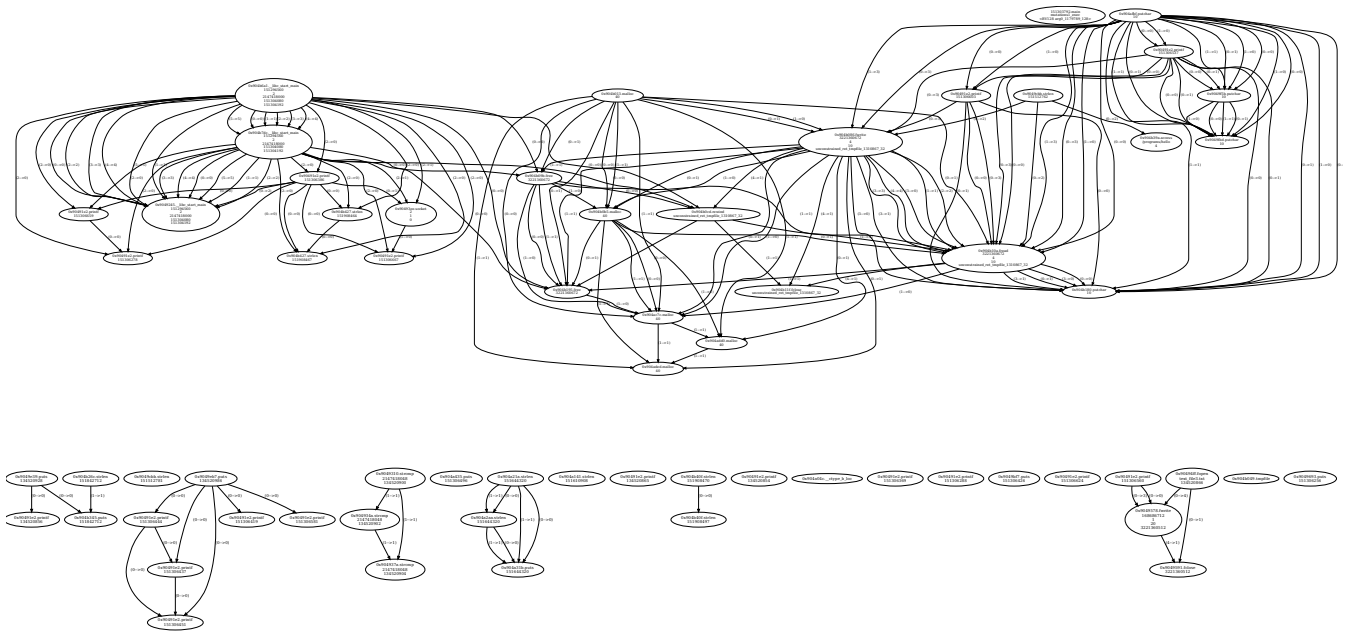
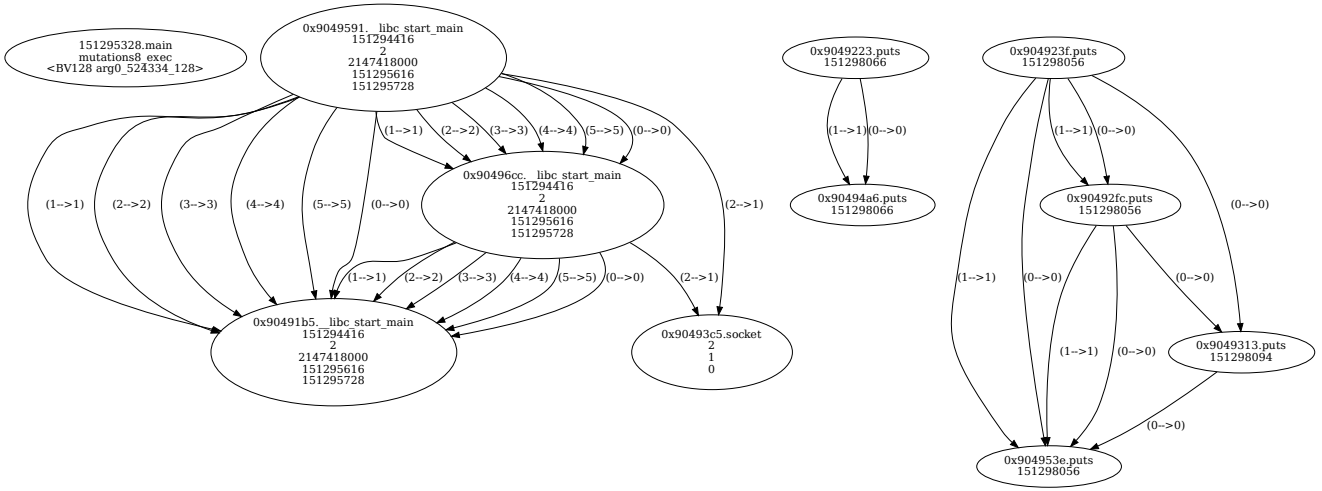


Figure C.1: SCDG after adding 18 random cleanware

C.2.2 Basic If



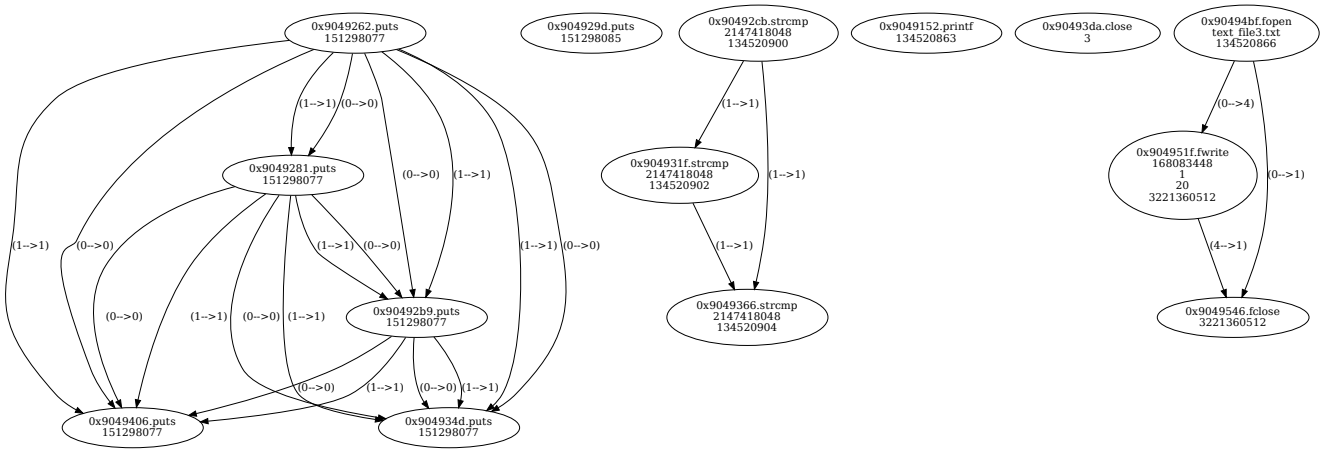
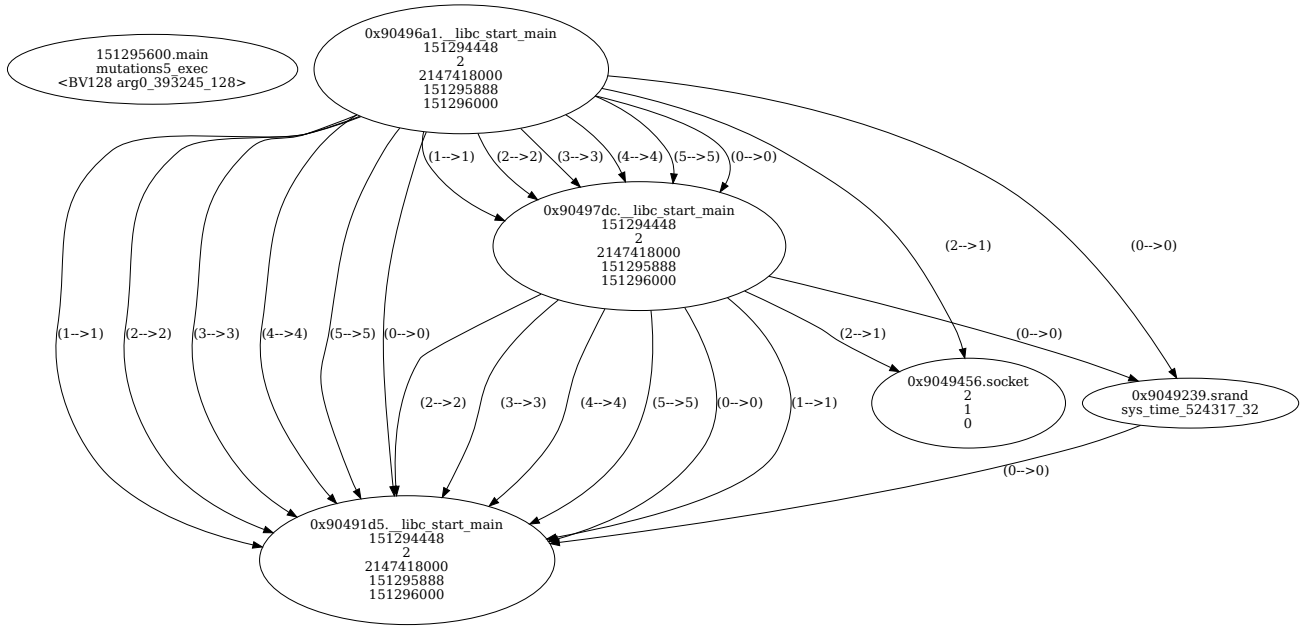


Figure C.2: SCDG after adding 12 basic if conditions

C.2.3 Random If



0x9049230.time
0x9049241.rand
0x9049260.rand
0x904926f.rand
0x904927e.rand
0x904928c.rand
0x904929b.rand
0x90492a6.rand
0x90492b5.rand
0x90492c4.rand
0x90492d3.rand
(0-->0)

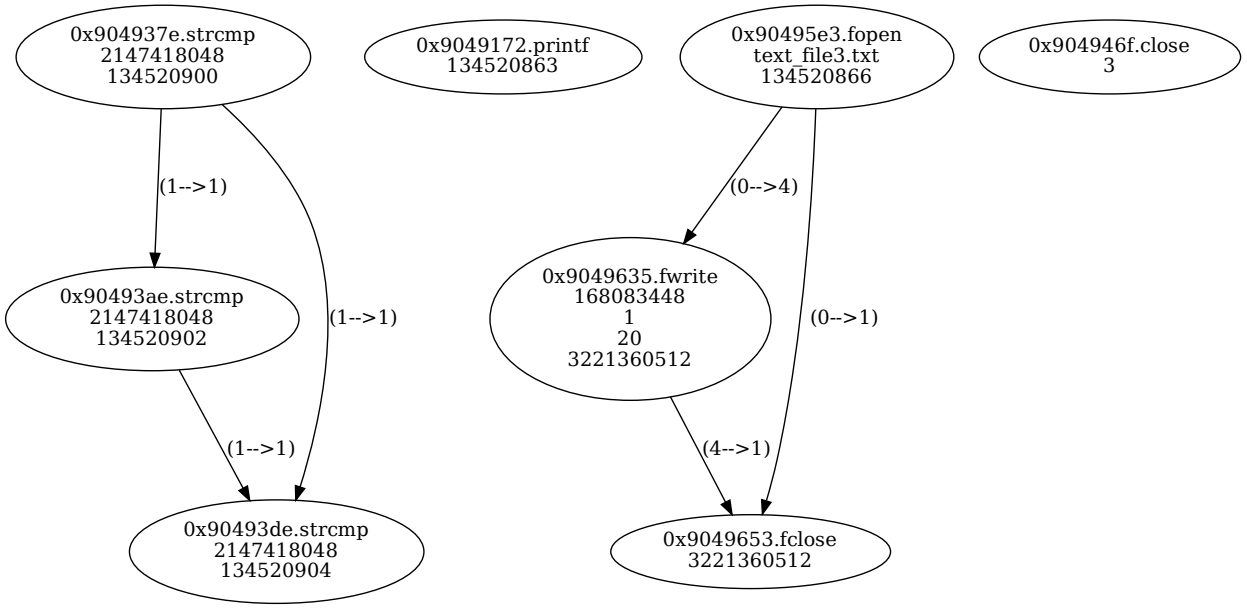


Figure C.3: SCDG after adding 10 random if conditions

C.2.4 System Call Adder

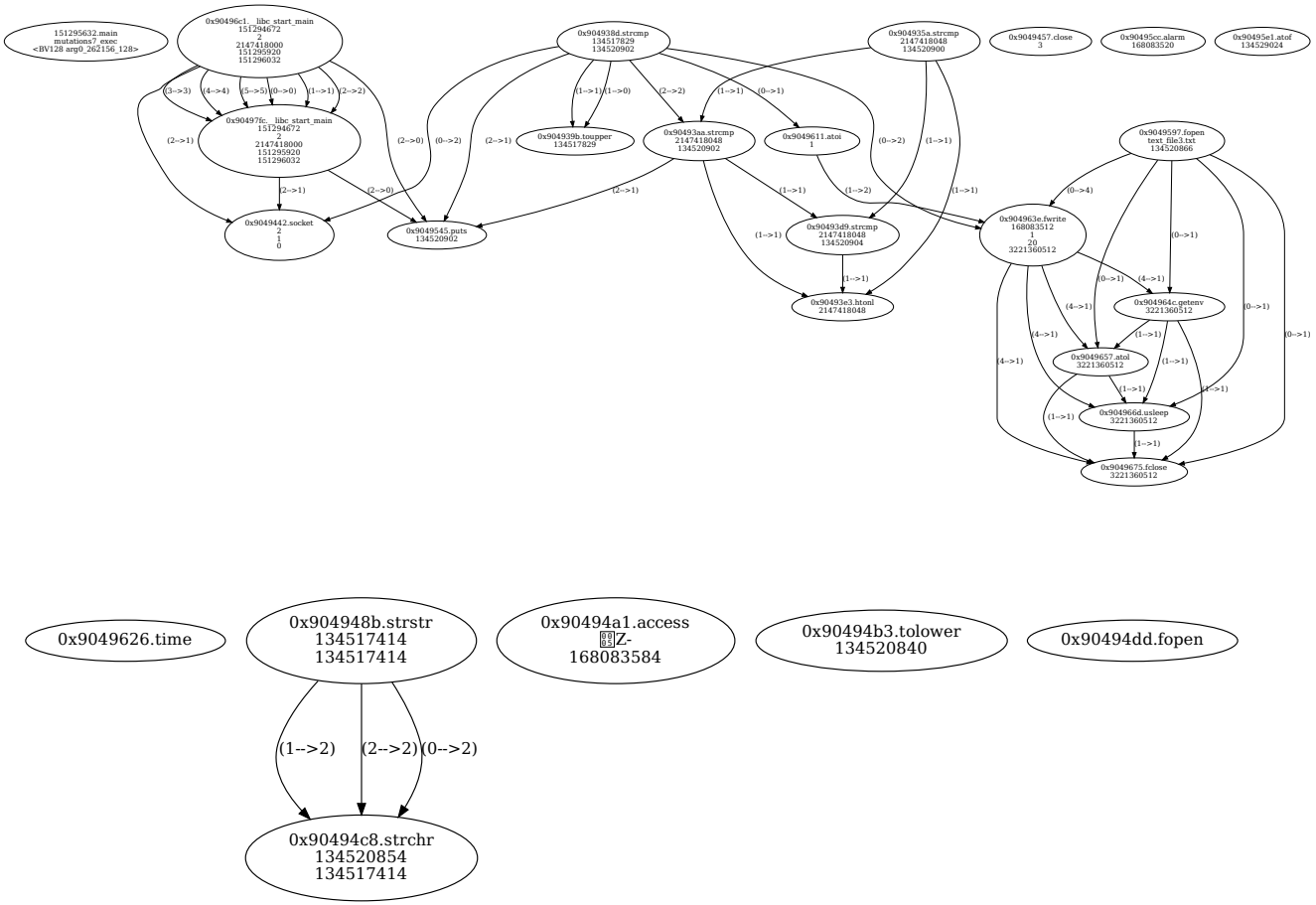


Figure C.4: SCDG after adding 19 system calls

C.2.5 Complete Mutation

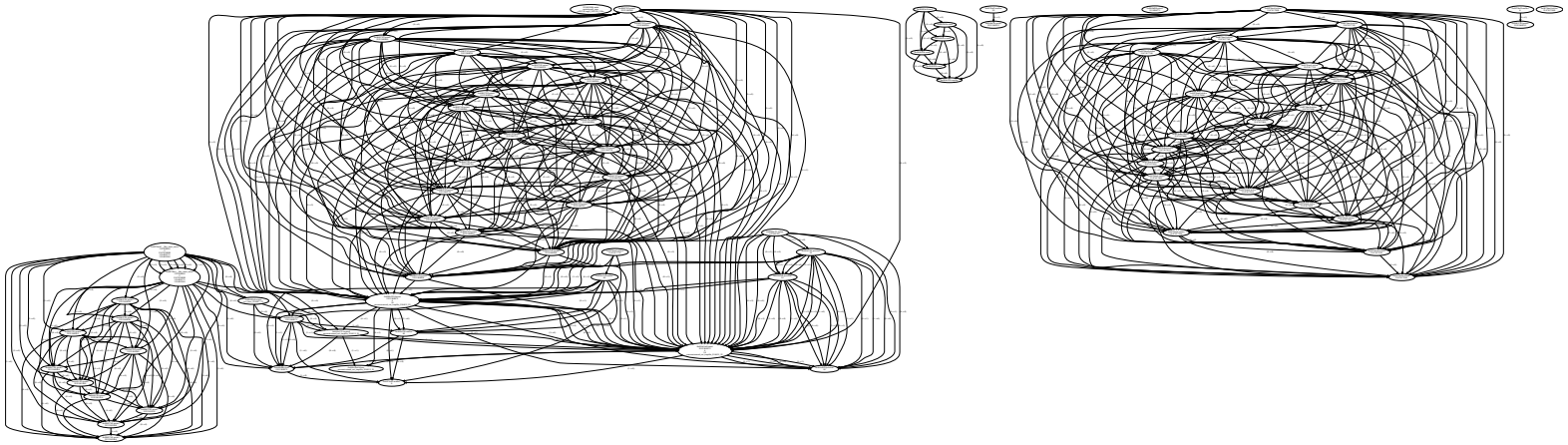


Figure C.5: SCDG after a combination of all mutations

C.3 Yara rules

C.3.1 Split rules

```
rule basic_string {
    strings:
        $evil = "I am evil"
    condition:
        $evil
}

rule splitted {
    strings:
        $evil = /I.{,16}am.{,16}evil/
    condition:
        all of them
}

rule extreme_split {
    strings:
        $evil = /I.{,16}a.{,16}m.{,16}e.{,16}v.{,16}i.{,16}l/
    condition:
        all of them
}

rule all_letters {
    strings:
        $I = "I"
        $a = "a"
        $m = "m"
        $e = "e"
        $v = "v"
        $i = "i"
        $l = "l"
        $space = " "
```

```
    condition:
        all of them
}
```

C.3.2 Encodings

```
rule XORed {
    strings:
        $xor = "I am evil" xor
        $xor1 = "I am evil" xor wide
        $xor2 = "I am evil" xor wide ascii
        $xor3 = "I am evil" xor ascii
    condition:
        any of them
}

rule Base64 {
    strings:
        $a = "I am evil" base64
    condition:
        $a
}
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**