Malware Analysis: A Perspective from Dynamic Symbolic Execution of Binary Code

DOI:https://doi.org/10.54654/isj.v2i25.1093

Mizuhito Ogawa

***Abstract*— Malware analysis typically involves three steps: obfuscation, infection, and malicious action. Many antivirus methods fail because obfuscation hides control structures. This paper provides an overview of dynamic symbolic execution (DSE) applied to binary code, especially x86. DSE is considered the most powerful technique for deobfuscation and can automatically recover control structures such as control-flow graphs. Several DSE tools target x86 (e.g., angr, Mayhem, S2E, KLEE-MC, and BE-PUM); we examine their design choices and trade-offs. Finally, we evaluate the effectiveness of control-flow graph similarity for tasks such as packer identification and original entry point (OEP) detection.**

***Tóm tắt*— Phân tích phần mềm độc hại bao gồm ba bước: làm rối, lây nhiễm và các hành vi độc hại. Nhiều phương pháp chống lại virus thất bại vì kỹ thuật làm rối che giấu các cấu trúc điều khiển. Bài báo này cung cấp cái nhìn tổng quan về thực thi biểu tượng động (DSE) được áp dụng cho mã nhị phân, đặc biệt với kiến trúc x86. DSE được xem là kỹ thuật mạnh mẽ nhất cho khử rối và có thể tự động khôi phục các cấu trúc điều khiển như đồ thị luồng điều khiển. Nhiều công cụ DSE mục tiêu x86 (ví dụ như angr, Mayhem, S2E, KLEE-MC và BE-PUM); chúng tôi xem xét các lựa chọn thiết kế và những đánh đổi của chúng. Cuối cùng, chúng tôi đánh giá hiệu quả của độ tương đồng đồ thị luồng điều khiển cho các tác vụ như nhận dạng trình đóng gói và phát hiện điểm vào ban đầu / nguyên thuỷ (OEP).**

Keywords*— *Dynamic symbolic execution, binary code, x86, control flow graph, obfuscation

Từ khóa*— *Thực thi biểu tượng động; mã nhị phân; x86; đồ thị luồng điều khiển; kỹ thuật làm rối.

I. Introduction

A control-flow graph (CFG) is one of the key features representing program behavior, as it reflects the underlying semantics of a program. For example, complementary to large language model (LLM) approaches [50], vulnerability detection in C/C++ programs often utilizes CFGs as part of the feature set, as seen in systems like VulDeePecker [47], Velvet [55], and DeepVD [51]. CFGs for C/C++ programs can be easily extracted through syntactic parsing. However, extracting CFGs from binary code is significantly more challenging. For instance, x86/Windows malware can employ mutation techniques [56] that alter the binary while preserving both the PE format and program semantics.

CFG generation from binary code is essentially equivalent to disassembly. Commercial disassemblers such as IDA Pro and Capstone perform syntactic analysis and work well on non-packed binaries - though they may fail to resolve indirect jump targets.

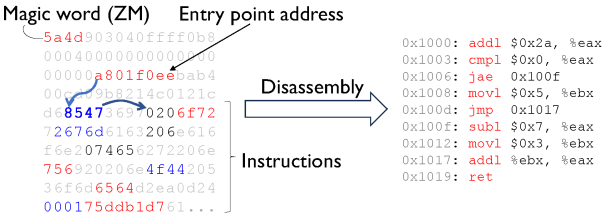

Figure 1. x86 binary and its disassembly

In contrast, packed binaries often use obfuscation techniques that easily defeat such syntactic disassembly. Alarmingly, more than 85% of modern malware is packed.

To address these challenges, Dynamic Symbolic Execution (DSE) is widely regarded as the most effective deobfuscation technique [57]. In addition to recovering control-flow structures,

This manuscript was received on March 11, 2025. It was reviewed on April 18, 2025, revised on April 30, 2025 and accepted on July 14, 2025.

DSE can also detect dead code by determining the infeasibility of execution paths. This capability is particularly valuable, as dead code can constitute up to 50% of the total code in x86/Windows malware.

This paper surveys dynamic symbolic execution (DSE) applied to binary code, with a particular focus on the x86 architecture. We also provide an overview of potential applications of DSE tools in binary analysis. Concepts are explained briefly and intuitively, with relevant references provided to guide interested readers.

This tutorial is organized as follows. Section II briefly mentions the obfuscation techniques to bypass anti-virus software and the infection techniques to attack vulnerability. Section III explains what is a CFG of binary code. Section IV reviews the existing DSE tools on binary code, especially x86. Section V discusses how DSE can traverse heterogeneous environments, like Android/apk. Finally, Section VI shows applications, e.g., packer identification, OEP (Original Entry Point) detection, and vulnerability detection.

II. Malware Techniques

Malware techniques consist of three steps: *obfuscation*, *infection*, and *malicious action*. You can refer the details of these techniques with code samples to [1 - 4].

A. Obfuscation techniques and their observation

Obfuscation techniques are classified in [11, 12]. Nowadays, most of them are automatically introduced by packers. By adding recent obfuscation techniques, e.g., vitualization, we categorize them.

1) Entry/code placing obfuscation (Code layout):
   overlapping functions, overlapping blocks, and code chunking.
2) Self-modification code (Dynamic code) : overwriting, packing/unpacking and SEH (structural exception handler)
3) Control flow obfuscation: Indirect jump and opaque predicate
4) Arithmetic operation: Obfuscated constants and checksumming.
5) Anti-debugging: Timing check, Special API, Stolen bytes and Dynamic loading.
6) Anti-tampering: Hardware breakpoint and anti-rewriting.
7) Virtualization: virtualization.

Apart from obfuscation, trigger-based behavior [14] is another technique used to evade dynamic analysis. Such triggers activate malicious actions only under rare environmental conditions (e.g., a specific time, date, or IP address), so dynamic analysis may fail to execute the malicious branches. In DSE, symbolic execution can check the satisfiability of these branches and generate concrete inputs that trace them.

Code layout

IA-32 (x86) uses a variable-length instruction set, so overlapping interpretations of a byte sequence are possible. For example, the byte sequences `b8 eb 07 b9 eb` and `0f 90 eb` can be decoded as `mov eax, ebb907eb` and `seto bl`, respectively; the fragment `eb 0f` can also be decoded as `jmp 45402c`. Such cases produce overlapping instructions. When overlapping instructions span function boundaries (resp. basic-block boundaries), we call them overlapping functions (resp. overlapping blocks).

When code is split into fragments that are connected only via jump instructions, we call this code chunking. In our work, we use a heuristic criterion: a region is considered chunked if there are three jump instructions within a window of 20 bytes or less [41].

Self-modification code

Overwriting refers to rewriting portions of the binary during execution. Packing/ unpacking is analogous to encryption/decryption: a payload is stored in encrypted form and dynamically decrypted by a runtime loop, whereas overwriting implies direct modification of code bytes. Packing is a major technique for evading antivirus software. For example, a polymorphic virus modifies its encryption key during infection, producing substantially different binaries even when the original payload is the same.

Control flow obfuscation

Indirect jumps conceal jump targets by computing and storing them in memory or registers at runtime, which hinders static resolution of control flow. An opaque predicate is a condition that is either always true or always

false; an opaque predicate in a conditional thus forces execution to follow a fixed branch, introducing dead code and confusing static analyses. For instance, `if` $x^2 < 0$ `...` and `if` $x > 2$ `then if` $x < 1$ `...` never choose the `then` branch. It has been reported that opaque predicates are responsible for a substantial fraction of dead code in contemporary malware.

Structured exception handling (SEH) provides user-level exception handlers and can be abused to implement indirect control transfers, further complicating control-flow recovery.

Arithmetic operation

Obfuscated constants hide argument values by replacing them with equivalent arithmetic expressions, e.g., $(x + 4) - 4$, or mixed boolean-arithmetic (MBA) encodings, making it difficult for static analyzers to recover the original constants.

Checksumming

Checksumming (e.g., CRC checksums) is used to detect whether a binary has been modified, for example when software interrupts are inserted for debugging. It is commonly employed as an anti-tampering mechanism.

Anti-debugging

Timing checks detect anomalies in execution time compared to a native Windows environment (for example, unusually slow execution). APIs such as `QueryPerformanceCounter`, `GetTickCount` are often used for this purpose. Special API checks, for example `CALL kernel32.IsDebuggerPresent` in `kernel32.dll`, probe whether the process is being debugged. Hardware breakpoints use the CPU debug registers ($DR0$ - $DR3$) rather than general-purpose registers (e.g., EAX, EBX, ECX) to monitor memory addresses; the `INT3` instruction is a software breakpoint that triggers a debug exception. The stolen-bytes technique allocates a buffer (typically with `VirtualAlloc`) and copies the unpacked code there instead of overwriting the original code region. Dynamic loading (e.g., via `LoadLibrary` and `GetProcAddress` in `kernel32.dll`) loads APIs dynamically and is also commonly used to hinder static analysis. There are many related techniques [2].

Anti-tampering

In addition to hardware breakpoints and `INT3` instructions, anti-rewriting techniques include stolen-bytes and checksumming to prevent or detect modifications.

Virtualization

Virtualization-based packers implement a custom virtual machine (VM) and translate original instructions into VM bytecode. Examples include `VMprotect`, `CodeVirtualizer`, `ExeCrypter`, and `Themida`. Some tools (e.g., `CodeVirtualizer`) even polymorphically modify their VM instruction sets. `VMProtect` is a typical example of a packer that combines virtualization with aggressive anti-debugging (consisting of multiple protection stages). Debugger plugins such as `ScyllaHide` (for `OllyDbg` and `x64dbg`) have been developed to defeat `VMProtect`'s anti-debugging measures; tools like `64unpack` can handle some versions, e.g., `VMProtect 3.4` [40].

B. Infection

Malware infection typically proceeds in three main steps; see [2 – 4] for details.

- Overwrite/leak the jump (return) destination by buffer overflow: Many attacks (over 90%in some studies) begin with a buffer-overflow vulnerability, often caused by unsafe memory operations such as `strcpy` or misuse of `printf` (omitting format specifiers in the first argument). These vulnerabilities lead to stack overflows (which commonly overwrite return addresses to redirect control flow) or heap overflows (which can corrupt heap metadata, file names, or other data structures).
- Set up malicious code: This can be done by invoking library APIs (Return-to-libc style attacks) or by injecting shellcode. In a Return-to-libc attack, the attacker overwrites a return address and arguments to jump to an existing API function in a library. Shellcode attacks place executable payloads in writable regions such as the data section or the stack (often as encoded strings) because many OSes mark the code section as read-only. Format-string vulnerabilities (e.g., careless `printf` usage that forgets the format specification in the first argument) can be

abused to write arbitrary values to memory. Naïve URL canonicalization that allows ".." sequences (directory traversal) or more sophisticated encodings (e.g., `UTF-8 conversion`) may be exploited; similarly, MIME encoding can be abused to smuggle payloads.

- Set up IAT to use APIs from shellcode: Once control is transferred to shellcode, the payload typically needs to call API functions. To do this, shellcode locates the image base of target DLLs (e.g., kernel32.dll, advapi32.dll, ws2_32.dll, sometimes gdi32.dll, user32.dll, ole32.dll), enumerates export names, and computes relative virtual addresses (RVAs) of required APIs. Implementations often use heuristic (e.g. assume that the offset would be a multiple of 1000) or iterative searches for these addresses.

III. Control flow graph of malware

A. Definition of control flow graph

The standard definition of a control-flow graph (CFG) is a directed, labeled graph in which a node represents a program location, an edge denotes a possible control transfer, and a label associates a node (or edge) with an expression. However, because binary code may be self-modifying, we represent a node as a pair consisting of a program location (address) and the instruction at that location; different instructions can be associated with the same address after overwriting. For example, Figure 2 illustrates the difference between these two representations when a jump destination is overwritten. This design is also effective for handling overlapping functions.

A basic block is defined as a maximal contiguous sequence of instructions with a single entry point and a single exit point - that is, it has no multiple incoming or outgoing edges. For example, the dynamic symbolic execution tool ANGR [33] constructs a control-flow graph (CFG) of binary code where each node represents a basic block, and each basic block is labeled with its corresponding instruction sequence.

B. Context sensitivity of CFG

CFGs were originally designed to represent intra-procedural control flow only - calls and returns are treated separately. When modeling inter-procedural control flow, such as function calls and returns, context sensitivity becomes important. In a context-sensitive setting, function returns must precisely return to the instruction immediately following the corresponding call site.

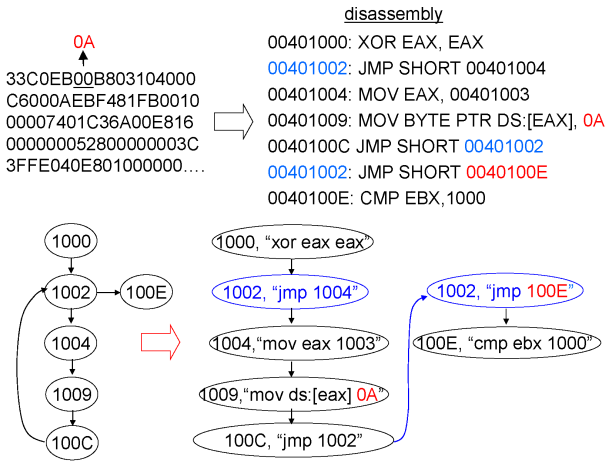

Figure 2. CFG for binary code

A call graph models relationships between functions, where nodes represent functions and edges represent calls from one function to another. However, conventional call graphs are context-insensitive, meaning that they do not distinguish between different call sites calling the same callee - returns may point to any of the callers indiscriminately.

Accurate context management requires simulating a call stack that records return addresses. However, standard CFGs are simply directed graphs and lack such stack structures. A common workaround is k-CFG [8], where callee functions are cloned for each call context up to k levels deep - this technique is known as *context cloning*. Note that since function calls can nest to arbitrary depth, unrestricted duplication may lead to infinite expansion, and we limit as k-CFG.

While more precise, k-CFGs can be computationally expensive; therefore, typical values are $k = 1$ or $k = 2$ (with $k = 0$ denoting a context-insensitive CFG). For example, `BE-PUM` (for x86/win) [1] adopts a 0-CFG approach, while `HybridSE` (for Android/apk on ARM)[2] uses a 1-CFG model as a practical choice between precision and complexity.

[1]https://github.com/NMHai/BE-PUM
[2]https://github.com/hybridse/HybridSE

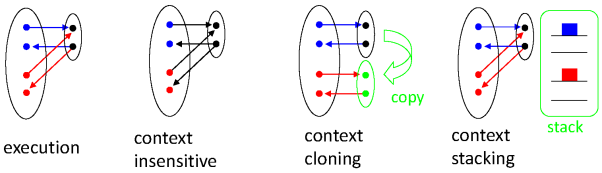

Figure 3. Context sensitivity of CFGs

C. Graph Kernel

The Weisfeiler-Lehman (WL) graph kernel computes a canonical feature vector for labeled graphs and is (almost) invariant under label-preserving isomorphisms [15, 16]. Intuitively, it extends the idea of n-grams from sequences to graphs by generalizing subsequences of length n to radius n subgraphs around each node. At each iteration of the WL algorithm, a new label is assigned to each node by hashing the multiset of labels in its neighborhood, as illustrated in Figure 4.

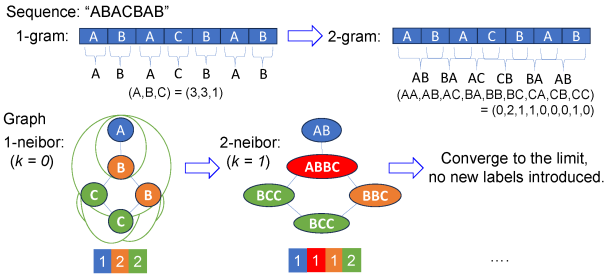

Figure 4. Intuitive idea of Weisfeiler-Lehman graph kernel

Repeating this process iteratively captures increasingly rich local structures. However, computing the WL kernel until convergence is computationally intensive.

There are several variants of graph kernels that serve as efficient approximations of graph similarity, including the shortest path kernel, random walk kernel, graphlet kernel, graph hopper kernel, and hash graph kernel [17]. These and other variations are implemented in the `GraKeL` library [3].

In addition, by combining graph kernel ideas with neural embeddings, tools such as `graph2vec` [4] have become popular for learning vector representations of graphs and computing graph similarity more effectively [18].

[3]https://ysig.github.io/GraKeL/
[4]https://github.com/benedekrozemberczki/graph2vec

IV. Dynamic Symbolic Execution on Binary Code

A. Hoare logic

Hoare logic is built on an underlying logic with the expression $\{A\}Program\{B\}$ that means if A holds then the execution of $Program$ concludes the postcondition B. (Detailes are referred in [7].) In Java, this speicication, described in Java Modeling Language (JML), can be a part of a program used in Design-by-contract and Assume-Guarantee approaches.

To see the idea, we will set a simple imperative program with while-loop. Its language constructs are:

- Assignment: $x = e$
- Conditional: $if\ C\ then\ S1\ else\ S2$
- Sequence: $S1; S2$
- While-loop: $while\ C\ do\ S$

where e is an arithmetic expression (i.e., $e = x \mid 0 \mid 1 \mid e + e \mid e \times e$, C is a condition constructed by predicates $=, <$, and S is a sequence.

Then, the deduction rules corresponding to each construct are:

$$\text{[Assignment:]}\{A[e/x]\}\ x = e\ \{A\}$$
$$\text{[Conditional:]}\{A\}\ if\ C\ then\ S1\ else\ S2\ \{B\}\{A \wedge C\}\ S1\ \{B\} \text{ and } \{A \wedge \neg C\}\ S2\ \{B\}$$
$$\text{[Sequence:]}\{A\}\ S1; S2\ \{C\}\{A\}\ S1\ \{B\} \text{ and } \{B\}\ S2\ \{C\}$$
$$\text{[While-loop:]}\{R\}\ while\ C\ do\ S\ \{R \wedge \neg C\}\{R \wedge C\}\ S\ \{R\}$$

Here R is called a *loop-invariant*, which holds before/after the execution of S. We have a inference rule

$$\text{[Consequence:]}\{A'\}\ S\ \{B'\} A' \supset A,\ B \supset B',\ \{A\}\ S\ \{B\}$$

Note that the inferences $A' \supset A$, $B \supset B'$ are not in Hoare logic, but in the underlying logic. Note that finding a loop invariant is a search to find R with $A \supset R \supset B$ in While-loop deduction rule. It is undecidable in general, but good search strategies are known, e.g., Farkas's Lemma and Craig interpolant.

In the context of symbolic execution, the choice of the underlying logic typically corresponds to the background theories supported by an SMT solver. For example, LIA (Linear Integer Arithmetic), UF (Uninterpreted Functions), BitVectors, and Arrays.

We call I of $\{A\}\ Prefix\ \{I\}$ a path condition of $Prefix$. Assume that we deduce $\{A\}\ Program\ \{B\}$. Then, if the program exit is reachable from the entry, the assumption A will

lead the conclusion B. Therefore, verifying such a program requires checking reachability, which is equivalent to proving termination. In this sense, program correctness is guaranteed only if termination is also established, which is known as partial correctness.

For extended programming features, such as pointers, context sensitivity, or higher-order functions, the limits of (relative) completeness of symbolic execution are discussed in [9, 10].

B. Dynamic symbolic execution

Concrete execution interprets a program over concrete (actual) values. In contrast, symbolic execution interprets a program over symbolic domains, where the program state is represented by symbolic formulae [19]. Starting from the entry point, symbolic execution updates the path condition at each step according to deduction rules, accumulating constraints that describe all feasible execution paths.

Figure 5 illustrates an example of symbolic execution on a small Java program. Initially, symbolic values α, β, γ are assigned to the input variables a, b, c, respectively. When the path condition at a particular program location is satisfiable, a concrete input that satisfies it can drive execution to that location. Conversely, if the path condition is unsatisfiable, the location is unreachable, i.e., it is considered dead code.

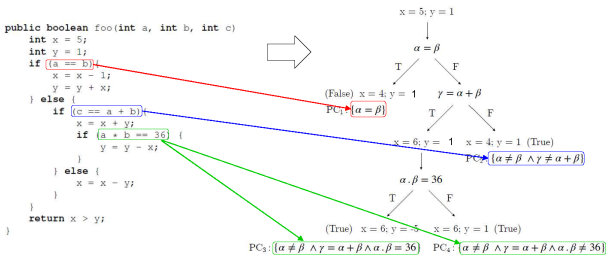

Figure 5. Example of symbolic execution on small java program

Symbolic execution is a classical technique [19], and practical implementations for programming languages have been evolving since the 1990s. Popular tools include `Klee` [22] for C programs, and SPF (Symbolic Path Finder) [23] and `JDart` [24] for Java bytecode.

These tools go beyond basic symbolic execution by supporting Dynamic Symbolic Execution (DSE) - also known as concolic testing. DSE combines traditional testing (concrete execution) with symbolic execution. Originally, DSE was proposed to improve efficiency by replacing parts of symbolic reasoning with concrete execution.

In malware analysis, this hybrid approach is especially useful for resolving indirect jump destinations, which are difficult to determine statically. While programming languages often make potential targets explicit through syntax, in obfuscated malware the jump targets are typically computed dynamically. Static analysis alone struggles to recover these due to obfuscations such as arithmetic manipulations.

Tools like McVeto [25] use weighted pushdown model checking to enumerate candidate jump targets. While effective on unpacked binaries, such methods often fail under real-world obfuscations. In contrast, DSE resolves jump targets by generating satisfying inputs for the path condition at the jump site and executing the binary with those inputs.

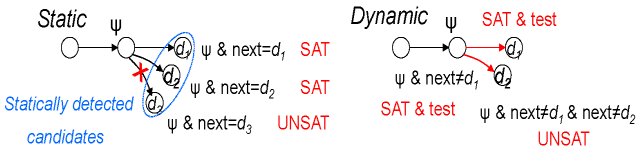

Figure 6. Finding indirect jump destinations by DSE

C. DSE on binary code

Popular DSE tools for x86 binary code include `ANGR` [33], `Mayhem` [29], `S2E` [28], `Klee-MC` [30], `BE-PUM` [31], `BINSEC/SE` [32]. Among them, `ANGR` is perhaps the most widely used, and supports multiple architectures beyond x86.

Many of these tools are built on intermediate machine languages (IMLs), which abstract away hardware-specific instruction sets. For instance:

- `ANGR` uses VEX,
- `Mayhem` uses BAP [26],
- `Klee-MC` uses LLVM, and
- `BINSEC/SE` uses DBA [27].

In contrast, BE-PUM directly interprets x86 binaries and performs DSE without translating to an IML. Each approach has its advantages and trade-offs:

- IML-based designs enable DSE to be applied across multiple architectures (e.g., x86, ARM, MIPS) by first translating the binary

to a common intermediate form. On the other hand, direct interpretation requires a dedicated implementation for each architecture (e.g., BE-PUM [31] for x86, Corana [34] for ARM, and SyMIPS [35] for MIPS). To reduce manual engineering effort, some tools use (semi-)automated extraction of formal semantics from instruction set architecture (ISA) manuals (e.g., x86 [45], ARM [34], MIPS [35].

- If IML translators are treated as black-box components, they inherit the limitations of static disassemblers, especially in the presence of obfuscation. For example, `ANGR`, `Mayhem`, `Klee-MC` rely on off-the-shelf IML translators. In contrast, BINSEC/SE integrates its IML (DBA) translator tightly with the symbolic execution engine. This step-by-step interaction enables BINSEC/SE to track dynamic changes more effectively, and to overcome limitations that purely static translators face under obfuscation.

An example architecture of a DSE tool is illustated in Figure 7. The example is `BE-PUM`.

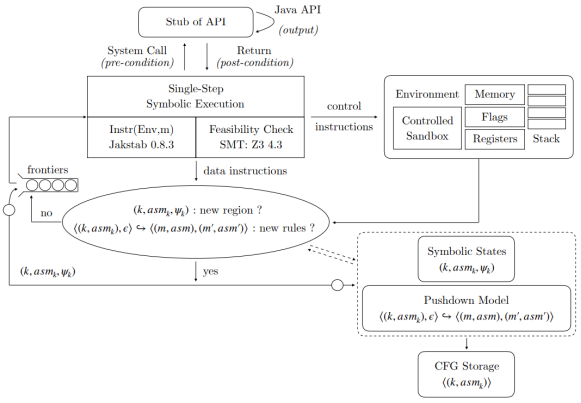

Figure 7. Architecture of DSE tool `BE-PUM`

V. DSE OVER HETEROGENEOUS ENVIRONMENTS

When applying DSE to x86 binary code, we frequently encounter Windows API calls or Linux library functions, which are essentially procedural calls to blackbox components. For such blackbox callees, the common approach is to instantiate the function arguments using satisfiable instances of the current path condition, and then perform concrete execution of the function.

In contrast, when analyzing an Android APK file, the execution spans multiple environments. An APK file is a packaged archive that can be decomposed using tools like `apktool`[5], which extracts components such as `AndroidManifest.xml`, `.dex` files, `.so` files (native code), and other resources. The `.dex` files contain Dalvik bytecode, a variant of Java bytecode, which can be converted into standard Java bytecode using `dex2jar`. The native code within the APK may be compiled for x86, ARM, or MIPS, though ARM is the most common target.

As a result, DSE on Android typically starts from the Java layer but may involve transitions to native code or Linux system calls. System functions are treated as blackboxes, while native code is typically analyzable. There are two strategies for handling native code during DSE:

- Blackbox treatment: Handle native code via concrete execution (i.e., do not symbolically analyze it).
- Whitebox treatment: Continue symbolic execution into the native code using a separate DSE tool designed for that architecture.

An example of the blackbox approach is `jpf-nhandler` [21], a plubin of `SPF` [23], which delegates native calls to concrete execution. Examples of the whitebox approach include `ANGR` [33] and `HybridSE`[6]. The former typically translates binary code (of X86, ARM, MIPS) into VEX IML; however, `ANGR` on Android/apk translates both Java bytecode and ARM native code into a unified intermediate representation, Jimple[7]. The latter connects SPF (for Java) with `Corana/API` [36] (for ARM native code), passing execution contexts between them while respecting the calling conventions.

A. Control passed to blackbox

When control is passed to a blackbox function (e.g., a native or system call), there are two common handling strategies in DSE:

- Over-approximation: Return new symbolic values as the result of the call.
- Under-approximation: Execute the call using a satisfiable concrete input.

We refer to the second strategy as concretization.

[5]https://apktool.org/
[6]https://github.com/hybridse/HybridSE
[7]https://soot-oss.github.io/SootUp/v1.1.2/jimple/

*O*ver-approximation is useful for detecting trigger-based behavior [14], such as time bombs (e.g., April Fool's Day logic) or sophisticated malware like *STUXNET*. However, excessive use of this approach can lead to state explosion, making DSE computationally intractable.

*C*oncretization, on the other hand, reduces symbolic execution to concrete execution at the blackbox boundary. It is particularly effective for analyzing system calls that search for dynamic resources such as files, IP addresses, or open ports. Two implementations of concretization are commonly found in practice:

- *F*ull Concretization of Execution Tools like `jpf-nhandler` (an `SPF` plugin) apply full concretization by maintaining parallel symbolic and concrete executions. When a blackbox call is encountered, the current concrete state of DSE is transferred to the base execution platform and executed concretely. Afterward, the result is propagated back into the symbolic environment.
- *M*inimal Concretization of Arguments Tools such as `BE-PUM` (for x86) and `HybridSE` (for Android/apk on ARM) adopt a minimal concretization strategy, where only the function arguments of the blackbox call are instantiated with satisfiable concrete values. The rest of the symbolic execution context (e.g., symbolic memory, other variables) remains untouched.
  Upon return from the concrete call, the symbolic environment is updated only with the returned value, while the path condition remains unchanged, i.e., the pre- and post-conditions of the call are assumed to be identical. This design minimizes the impact on the symbolic state, as illustrated in Figure 8.

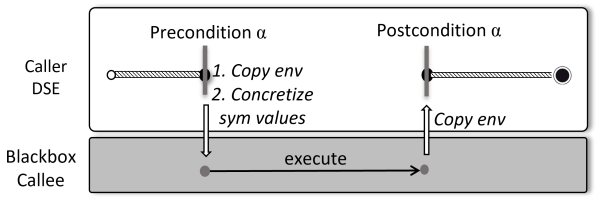

Figure 8. Partial concretization at blackbox call

B. Control passed to whitebox

When handling whitebox calls, there are two primary approaches:

- *C*onvert the Entire Program into a Single Context The standard implementation, ANGR, uniformly converts input programs into its intermediate representation (IML), VEX. Different from the standard traslation of `ANGE`, `ANGR` converts both Java bytecode and ARM native code in an Android/APK into Jimple, a common Java IR. However, this unified conversion introduces new challenges, especially semantic mismatches: Jimple preserves Java's object-oriented structure, including class hierarchies. Native code, in contrast, lacks this structure and operates under different memory and execution models. This mismatch makes semantic translation non-trivial and can affect the accuracy of symbolic execution.
- *C*ombine DSE Tools for Individual Platforms `HybridSE` takes a different approach by combining two distinct DSE engines: `SPF` for Java and `Corana/API` for ARM native. At function call and return boundaries, environment transfer is performed according to the calling conventions. This includes: (1) Copying argument values, (2) Mapping memory regions, and (3) Maintaining consistent execution state. Fig. 9 llustrates the control flow between two DSE tools during whitebox call transitions.

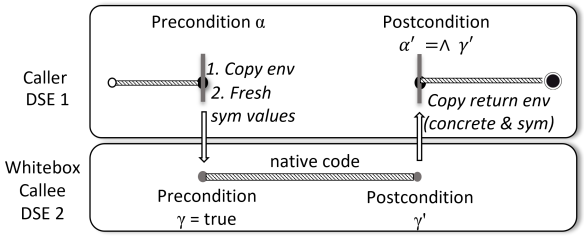

Figure 9. Traverse two DSE tools at whitebox call

Type-Sensitive Environment Transfer

Transferring the environment across different execution platforms requires careful handling of data types:

- For primitive types, copying the value into a register or pushing it onto the stack is sufficient (depending on the calling convention).

- For pointers or fields, memory tracing is required, since the referenced memory may represent: single value (e.g., int*), fixed-size buffer, and dynamically sized array terminated by a sentinel. Figure 10 illustrates the complexity of handling different pointer-based data structures across symbolic execution boundaries.

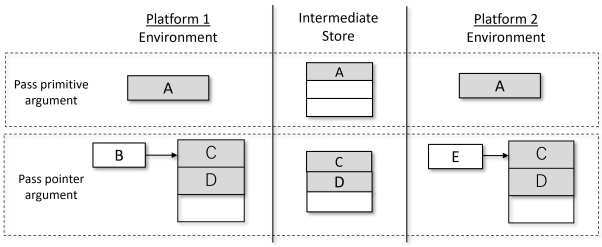

Figure 10. Passing environments between two different DSEs

Handling Windows API Calls

Windows APIs are vast, with thousands of documented functions. Although only around 1,000 are frequently used in practice, manually checking argument types from documentation remains a significant engineering effort.

To mitigate this:

- Naming conventions of arguments are used as weak type hints.
- Lightweight machine learning techniques are applied to infer pointer types and argument semantics.

With these methods, 60–70% of pointer-type arguments can be automatically inferred from OS manuals (for x86/Windows [44]) and Library code (for Android/apk [36]).

VI. Applications of CFG similarity

We now explore applications of CFG similarity, primarily using the Weisfeiler-Lehman graph kernel as the base method. In practice, we employ `grah2vec` [18], a neural embedding approach for graph similarity.

Most modern malware is packed using a variety of packers. Thus, a typical malware sample can be considered as a pair of the original payload and the packer used. To classify malware meaningfully, we must:

- Extract and compare CFGs of the original payloads, not directly the packed binaries.
- Accurately locate the Original Entry Point (OEP) of execution.

Figure 11 illustrates the working of a packer and its unpacking stub. For more technical details about packers, see references [11–13]. *(General) unpacking* refers to the process of locating the OEP and recovering the original binary payload.

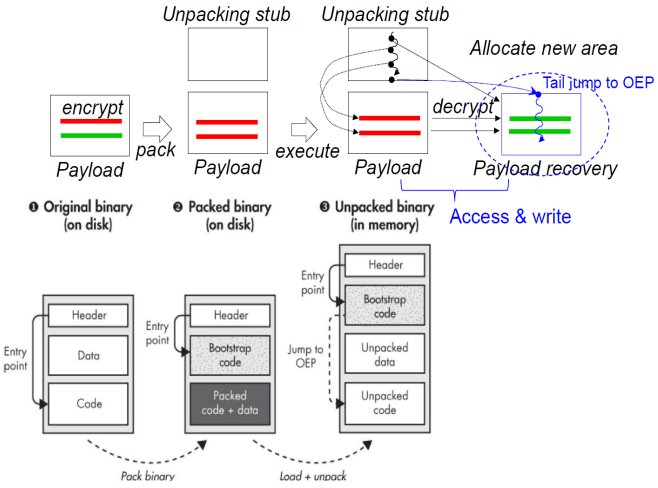

Figure 11. Packing and the unpacking stub by a packer

The packer typically (1) Encrypts the payload, (2) Adds an unpacking stub, and (3) Transfers control to the OEP after decryption. Applications introduced here are:

- Packer identification [41, 42]
- OEP (Original Entry Point) detection [42]
- Vulnerability detection [46]

A. Packer identification

As shown in [41], each packer introduces a characteristic sequence of obfuscation techniques within its unpacking stub.

In the experiment, four toy assembly programs (without loops or obfuscations) were packed using 12 different packers, including: ASPACK v2, CEXE v1.0b, FSG v2.0, KKRUNCHY v0.23a4, MPRESS v2.19, NPACK v1.0, PECOMPACT v2.0x, PETITE v2.1, TELOCK v0.99, UPX v3.0, YODA's Crypter v1.3, and UPACK v0.37-0.39. The DSE tool BE-PUM was applied to extract the CFGs of the unpacking stubs.

Based on CFG classification introduced in Section I, each packer showed a distinct structural

TABLE I. LIST OF OBFUSCATION TECHNIQUES

0	*overlapping function*	1	*overlapping block*	2	*code chunking*
3	*overwriting*	4	*packing/unpacking*	5	*indirect jump*
6	*SEH*	7	*2API*	8	*obfuscated constant*
9	*checksumming*	10	*timing check*	11	*anti-debugging*
12	*stolen bytes*	13	*hardware break point*		

ASPack v2.12	8_3_3_3_3_7_3_3_5_12_3_4_5_12_3_4_4_8_4_4_4_4_8_8_8_4_4_4_4_4_4_4_4_8_8_8_8_4_8_8_4_8_4_4_3_4_3_5_3_3_3_7_3_3_5_3_7
CEXE	5_4_4_5_4_4_4_4_4_6_8_4_4_6_8_4_4_4_8_5_5_7_4_4_4_4_4_4_4_8_3_3_4_3_3
FSG v2.0	3_3_5_3_5_4_3_5_3_5_3_4_3_5_12
KKRUNCHY	4_5_8_8_4_4_4_5_4_4_4_5_5_4_4_5_5_7_4_8_4_4_4_4_4_2_4_8_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4
MPRESS	4_4_4_8_8_8_8_8_8_4_4_4_4_4_4_2_4_8_3_3_7_5_5_3_4_3_4_3_3_3_3_3_3_4_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3_3
nPack v1.0	3_12_5_7_12_4_4_12_3_4_3_3_4_8_3_12
PECompact 2.0x	6_3_3_3_13_5_12_3_3_3_4_4_4_5_8_4_3_5_12_5_12_4_4_3_3_4_4_4_4_4_4_4_4_4_4_8_3_3_5_3_12_5_3_5_4_5_3_3_3
PEtite v2.1	3_3_3_3_3_4_4_3_4_2_8_8_8_8_8_8_8_4_4_4_4_8_4_4_8_4_8_4_4_4_4_4_4_4_3_9_5_8_4_4_3_6_8_4_4_8_8_8_3_3_3_3_3_1_5_3_7_4_3_3_4_
tElock 0.99	4_3_6_6_8_8_8_2_8_2_4_4_4_3_2_5_3_2_4_6_13_6_6_6_6_6_8_8_8_4_6_2_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_4_3_4_3_5_3_4_4_6_6_6_6_2_6_6_4_5_3
UPX v3.94	4_4_4_8_4_5_3_12_5_3
yoda's Crypter 1.3	8_2_3_4_3_8_9_3_3_6_8_13_8_8_3_3_3_3_3_7_3_3_3_3_3_3_3_3_3_3_5_3_5_4_5_3_3_3_5_2_3_4_4_5_3_3_3_4_5_7_2_4_4_4_8_8_3_4_3_12_5_0_3_6_8_13_8_3_4_8_3_4_6_8_13_12
UPACK v0.37	4_4_4_3_8_4_5_3_8_3_4_3_5_3_5_3_5_3_5_3_4_3_4_3_5_4_3_5_8_3_3_5_12

Figure 12. Obfuscation technique sequences for the observed packers

pattern. The results are summarized in Table I.

The detected obfuscation technique sequences are uniform for each packer. The results are summerized in Figure 12.

In [41], instead of using the exact sequence of obfuscation techniques as a signature, a frequency vector representing the number of occurrences of each technique was used. This vector is referred to as a *metadata signature*.

This method was applied to a large dataset of 12,814 real-world malware samples, including 5,374 samples from VX Heaven and 7,440 samples from VirusShare. Using the DSE tool BE-PUM, CFGs were successfully generated for 12,315 of these samples.

The identified packers were then compared with the results from widely used site `VirusTotal`[8] and tools `CFFexplorer`[9], `PEiD`[10].

Among the four tools (including BE-PUM), the inconsistency is on 933 samples:

- 325 samples were reported as "unknown" only by BE-PUM, likely due to the use of other existing packers not included among the 12 packers above.
- 206 samples were classified as "packed" by BE-PUM, but reported as "unpacked" by VirusTotal, CFF Explorer, and PEiD, likely due to the use of custom-made packers.
- 402 samples were inconsistent among the tools.

After manual analysis, BE-PUM (and PEiD) were found to be incorrect in only one case, where MEW was mistakenly identified as FSG [41].

In a follow-up study [42], instead of using metadata signatures, CFG similarity was used to identify the packer, along with OEP (Original Entry Point) detection. This approach is based on the observation that the unpacking stub - and thus the characteristics of the packer - can be identified near the OEP.

B. OEP detection

Most traditional methods for OEP detection rely on heuristics to hook the end of the unpacking stub [37–40]. For example, when decrypting an encrypted payload, an unpacking stub may:

- Allocate a new buffer in memory,
- Write the decrypted payload into that buffer, and
- Use a jump or return instruction to transfer control to the OEP.

Heuristics are based on detecting patterns in memory allocation, memory access, stack usage, and jumps. However, such heuristics are often packer-specific, limiting their generality.

Instead, [42] investigates a more robust and general approach to OEP detection using CFG similarity. The core idea is,

- When the packer is identified by the similarity of the unpacking stubs, the packed code execution is around OEP.
- Look for the code sequence that has the same tail-jump sequence around there.

assuming that the unpacking stub produced by a packer is uniform, regardless of the original payload.

For supporting the idea, we first apply a known packer (even as a black box) to various known binaries to collect (1) CFGs of unpacking stubs and (2) Tail-jump sequences (typically 5 instructions at the end of the stub).

Figure 13 shows the CFGs of unpacking stubs for UPX, FSG, and MEW. On the left, two binaries packed with UPX show that the CFGs of the unpacking stubs have nearly identical shapes, despite differences in the original payloads. On

[8]https://www.virustotal.com/
[9]https://ntcore.com/explorer-suite/
[10]https://github.com/wolfram77web/app-peid

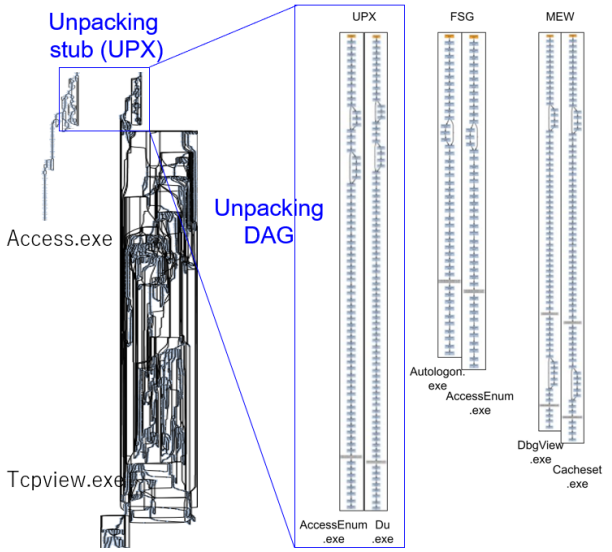

Figure 13. CFGs of unpacking stubs: UPX, FSG, and MEW

the right, the unpacking stubs of UPX, FSG, and MEW are shown for comparison.

The tail-jump sequence is also characteristic for each packer. For example, UPX commonly uses the following instruction sequence `pushl cmpl jne subl jmp`.

From these observations, the procedure proceeds:

- When a CFG of an unpacking stub matches a known stub, the corresponding packer is identified.
- The method then searches for a matching tail-jump sequence near the end of the stub.
- If such a match is found, the jump target is assumed to be the OEP.

This technique offers a systematic and generalizable approach to both packer identification and OEP detection, even in the absence of traditional heuristics.

In [42], packer identification and OEP (Original Entry Point) detection were evaluated on a dataset consisting of:

- 771 samples collected from GitHub repositories[11][12], and
- 1,259 malware samples from VXHeaven (a subset of the dataset used in [41]).

Among the former, 71 samples (covering 12 packers used in [41]) were used as *templates* representing the control flow graphs (CFGs) of the unpacking stubs. First, BE-PUM was used to generate CFGs from the dataset. Then, graph similarity was computed using `graph2vec`, which provides an approximation of the Weisfeiler-Lehman graph kernel. The results were compared against the following tools, (1) `VirusTotal`, (2) `PyPackerDetect`[13], (3) `Metadata signature` [41] (a frequency-based method) (4) `Gunpacker v0.5`[14], and (5) `QuickUnpack v2.2`[15]. The first three are for packer identification and the latter two are for OEP detection. Table II) shows the result, in which the column CFG presents our method.

Two failure cases were observed:

- WinUpack: WinUpack has two different unpacking stub templates (see Fig. 14). However, [42] used only one template (WINPACK1), likely due to the limited size of the training set.
- Packman: In one failed sample, the unpacking stub was executed twice, which was not accounted for in the detection logic.

Despite these failures, our CFG similarity approach for unpacking stub identification performed well, although it assumes access to the packer to prepare templates.

VII. Future applications of CFG similarity: Vulnerability detection

Detecting vulnerabilities in C/C++ programs has become a highly active area of research using machine learning (ML) and deep learning (DL). While these methods often apply learning on graph structures, the graphs are not always CFGs. Common graph-based representations include:

- Code Property Graph: Combines syntax, control flow, and data dependency [43].
- Call dataflow slice: Used in VulDeePecker [47]
- Def-Use chain: GraphCodeBERT [48]
- Code as Sequence + n-gram Co-occurrence: ReGVD [49]
- n-hop neiborhood graphs (control / data flow): LineVD [50]
- Combining Structural Graphs: DeepVD [51] combines control flow, data flow, exception

[11]https://github.com/chesvectain/PackingData
[12]https://github.com/packing-box/dataset-packed-pe
[13]ttps://gitub.com/cylance/PyPackerDetect
[14]https://webscene.ir/tools/show/GUnPacker-v0.5
[15]https://www.aldeid.com/wiki/QuickUnpack

TABLE II. COMPARISON WITH EXISTING TOOLS

Packer	Samples	Packer Identification				OEP detection		
		(1)	(2)	(3)	**CFG**	(4)	(5)	**CFG**
UPX v3.95	85	85	30	84	85	78	85	85
ASPACK v2.12	68	68	68	68	68	56	68	68
FSG v1.0	75	75	75	75	75	70	75	75
PECOMPACT v2.xx	27	27	27	27	27	0	8	27
MEW SE v1.2	75	75	75	75	75	74	8	75
YODA's Cryptor v1.3	74	74	74	62	74	73	8	74
PETITE v2.1	34	34	34	34	34	0	8	34
WINUPACK v.039 final	26	26	26	26	15	26	4	15
MPRESS v2.xx	78	78	0	78	78	0	8	78
PACKMAN v1.0	79	79	79	0	79	79	8	78
JDPACK v1.01	52	51	0	0	52	45	2	52
TELOCK v0.98	27	27	27	27	27	24	1	27
Total	700	699	515	556	689	525	283	688

flow, and call graphs. 6 properties, e.g., control flow, dataflow, exception flow, and call graph.
- Code Transformation Graphs: CodeJIT [52]

These techniques focus on function-level vulnerability detection. More fine-grained techniques, like line-level localization, aim to identify vulnerabilities within a small number of code lines (e.g., 10 lines). Such examples include:

- IVDetect [53]: Applies GCNN on neighborhood graphs.
- Velvet [55]: Uses GCNN on Code Property Graphs.
- LineVul [54]: Uses 2-gram statements with self-attension.

Obtaining CFGs from C/C++ programs is generally straightforward. However, when we consider the vulnerability detection on binary code, e.g., x86, we face the problem how to obtain CFGs of binary code. If not packed compiled code (like most of victim code), often popular disassemblers work. However, malware often contains heavy obfuscation, making CFG extraction difficult. In such cases, Dynamic Symbolic Execution (DSE) is considered one of the most powerful deobfuscation tools available.

VIII. LIMITATIONS OF DSE

Although DSE is considered a powerful deobfuscation method, it has several significant limitations [59].

- VM awareness / Anti-VM Techniques: Malware often detects whether it is running inside a virtualized environment using anti-VM techniques. These exploit implementation-specific behaviors that are difficult to cover fully. This is a cat-and-mouse game: as one method is mitigated, new detection techniques emerge. BE-PUM, like most DSE tools, fails to execute binaries packed with VMProtect, whereas `ScyllaHide` bypasses anti-debugging in VMProtect 3.0 and `x64unpack` circumvents VMProtect 3.4 [40].
- Path explosion: Originally, DSE (also known as concolic testing [20]) was intended to improve efficiency by combining symbolic and concrete execution. However, in malware analysis, the purpose shifts: instead of efficiency, it is used to resolve indirect jumps. When an indirect jump is encountered, DSE generates a satisfiable input that reaches the jump. It executes the jump concretely and finds the indirect jump destination. Repeating this process, DSE can identify possible jump targets.
  However, if obfuscation adds excessive conditional branches, DSE can become computationally infeasible due to path explosion [60, 61].
- Difficult or unsolvable symbolic expression: DSE relies on SMT solvers to evaluate path conditions. These solvers support backend theories like Arithmetic (linear/nonlinear), Uninterpreted functions, Bit-vectors, and Arrays. Problems arise when the symbolic expression is too complex or unsolvable, or

data structures (like arrays, lists, and structs) require modeling all elements symbolically. Approximations are often required in such cases [58, 59, 62].

- Virtualization and ROP: In virtualized obfuscation, the malware uses a custom instruction set, and understanding the control flow requires interpreting these instructions. Return-Oriented Programming (ROP) further complicates analysis by replacing jump instructions with sequences of push and return instructions. Several solutions were proposed like trace normalization via DSE and dynamic analysis and CFG reconstruction. They adapt Equational rewriting [63], Rule-based simplification [64, 66, 68], ML-based synthesis [67], and Time-stamped execution traces [65]. Even with virtualization, the malware ultimately executes on the original instruction set, although the control structure is modified. Another approach is to abstract the CFG, ignoring small variations but preserving the high-level structure. These abstracted CFGs can then be used in statistical or ML-based analysis.

IX. Conclusion

This paper reviewed:

- The fundamentals and tools of Dynamic Symbolic Execution (DSE) on binary code.
- Applications such as malware unpacking, CFG similarity, and vulnerability detection.
- Limitations of DSE in real-world scenarios.

While DSE is a powerful tool, it is computationally expensive and not suitable for real-time detection. However, it remains crucial for deep understanding of malware, particularly in revealing *Semantics*, *Unpacking behavior*, and *Code-level attack techniques*. Current applications focus on generating control flow graphs (CFGs). The next frontier lies in classifying malware and understanding its techniques, especially those targeting known vulnerabilities.

X. Acknowledgement

The author thanks to Prof. Jean-Yves Marion (University of Lorraine) and Dr. Nguyen Van Anh (Yokohama National University) for continuous collaboration. This research was supported by JSPS KAKENHI Grant Number 20K20625.

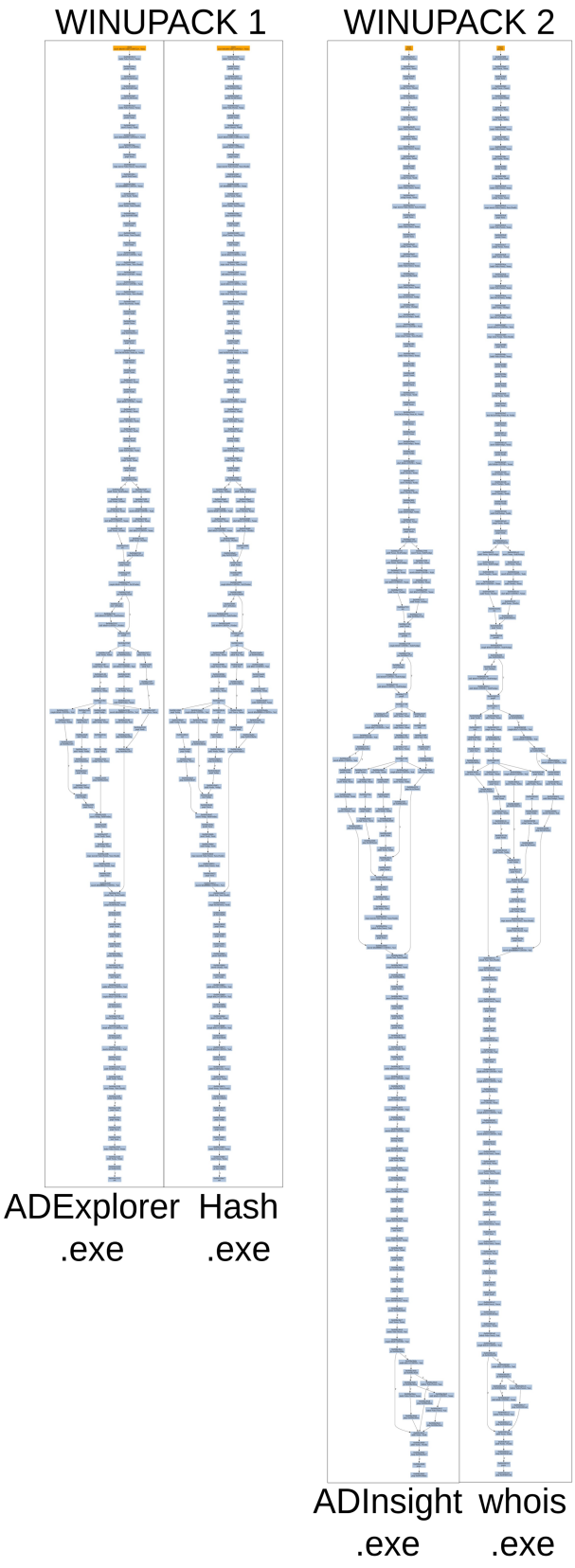

Figure 14. Two types of CFGs of unpacking stub of WINUPACK

REFERENCES

[1] M.Sikorski, A.Honig: Practical Malware Analysis, No Stretch Book, 2012.

[2] D.Andriesse, Practical Binary Analysis, No Stretch Book, 2018.

[3] P.Szor: The Art of Computer Virus Research and Defense, Addison Wesley, 2005

[4] C.Anley, J.Heasman, F.Lindner, G.Richarte: The Shellcoder's Handbook (2nd ed), Addison Wesley, 2007

[5] C.Collberg, J.Nagra: Surreptitious software. Addison Wesley 2010

[6] Nu1L Team: Handbook of CTFer, Springer, 2022.

[7] G.Winskel: The Formal Semantics of Programming Languages, MIT Press, 1993.

[8] F.Nielson, H.R.Nielson, C.Hankin: Principles of Program Analysis, Springer, 1999

[9] E.Clarke: Programming language constructs for which it is impossible to obtain good Hoare axiom systems, JACM 26(1), 1979.

[10] E.Clarke, S.M.German, J.Y.Halpern: Effective axiomatizations of Hoare logics, JACM 30(3), 1983.

[11] K.A.Roundy, B.P.Miller: Binary-code obfuscations in prevalent packer tools. ACM Comput. Surv 46 4:1-4:32, 2013

[12] S.Schrittwieser: Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? ACM Comp Surv 49(1), 2016.

[13] B.Cheng, J.Ming, E.A.Leal, H.Zhang, J.Fu, G.Peng, J.-Y.Marion: Obfuscation-Resilient Exectuable Payload Extraction From Packed Malware, *USENIX*, 3456, 2021.

[14] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, H. Yin: Automatically identifying trigger-based behavior in malware, *Botnet Detection* ADIS 36, 2008.

[15] B.Weisfeiler, A.A.Lehman: A reduction of a graph to a canonical form and an algebra arising during this reduction. Nauchno-Technicheskaya Informatsia 2(9), 1968.

[16] N.Shervashidze, P.Schweitzer, E.J.van Leeuwen, K.Mehlhorn, K.M.Borgwardt: Weisfeiler-Lehman Graph Kernels. Journal of Machine Learning Research 12, 2011.

[17] N.M.Kriege, F.D.Johansson, C.Morris: A survey on graph kernels, Applied Network Science 5:6, 2020.

[18] A.Narayanan, M.Chandramohan, R.Venkatesan, L.Chen, Y.Liu, S.Jaiswal: Graph2vec: Learning Distributed Representations of Graphs, https://arxiv.org/pdf/1707.05005, 2017.

[19] J.C.King: Symbolic execution and program testing, *Commun. ACM*, 19(7), 1976.

[20] P.Godefroid, N.Klarlund, K.Sen: DART: directed automated random testing, *PLDI*, 2005.

[21] N. Shafiei, F. van Breugel: Automatic handling of native methods in java pathfinder, *SPIN*, 2014

[22] C. Cadar, D. Dunbar, and D. Engler: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," *OSDI*, 2008.

[23] C. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, N. Rungta: Symbolic pathfinder: Integrating symbolic execution with model checking for java bytecode analysis, *ASE*, 2013

[24] M.Mues, F.Howar: JDart: Dynamic Symbolic Execution for Java Bytecode, *TACAS*, LNCS 12079, 2020

[25] A.V.Thakur, J.Lim, A.Lal, A.Burton, E.Driscoll, M.Elder, T.Andersen, T.Reps: Directed Proof Generation for Machine Code, *CAV*, LNCS 6174, 2010

[26] D.Brumley, I.Jager, T.Avgerinos, E.J.Schwart: BAP: A Binary Analysis Platform, *CAV*, LNCS 6806, 2011

[27] A.Djoudi, S.Bardin: BINSEC: Binary Code Analysis with Low Level Regions, *TACAS*, Springer LNCS 9035, 2015

[28] V. Chipounov, V. Kuznetsov, G. Candea: S2E: A platform for in-vivo multi-path analysis of software systems, *SIGARCH Comput. Archit. News* 1, 2011

[29] S. K. Cha, T. Avgerinos, A. Rebert, D. Brumley: Unleashing Mayhem on binary code, *SP*, 2012

[30] A. Romano: Methods for binary symbolic execution, PhD Dissertation, Stanford University, 2014.

[31] M.H.Nguyen, M.Ogawa, Q.T.Tho: Obfuscation code localization based on CFG generation of malware, *FPS*, LNCS 9482, 2015.

[32] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, J. Marion: BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis, *SANER*, 2016.

[33] Y.Shoshitaishvili, R.Wang, C.Salls, N.Stephens, M.Polino, A.Dutcher, J.Grosen, S.Feng, C.Hauser, C.Kruegel, G.Vigna: (State of) The Art of War: Offensive Techniques in Binary Analysis, *SP*, 2016.

[34] A. Vu, M. Ogawa: Formal semantics extraction from natural language specifications for ARM, *FM*, LNCS 11800, 2019.

[35] Q.T.Trac, M.Ogawa: Formal Semantics Extraction from MIPS Instruction Manual, FTSCS, Springer CCIS 1165, 2019.

[36] A. T. V. Nguyen, M. Ogawa: Automatic stub generation for dynamic symbolic execution of arm binary, SoICT, 2022.

[37] P.Royal, M.Halpin, D.Dagon, R.Edmonds, W.Lee: Automating the Hidden-Code Extraction of Unpack-Executing Malware, *ACSAC*, 2006.

[38] L.Martignoni, M.Christodorescu, S.Jha: OmniUnpack: Fast, Generic, and Safe Unpacking of Malware *ACSAC*, 2007.

[39] B.Cheng, J.Ming, J.Fu, G.Peng, T.Chen, X.Zhang, J.-Y.Marion: Towards Paving the Way for Large-Scale Windows Malware Analysis: Generic Binary Unpacking with Orders-of-Magnitude Performance Boost, *CCS*, 2018.

[40] S.Choi, T.Changi, C.Kim, Y.Park: x64Unpack: Hybrid Emulation Unpacker for 64-bit Windows Environments and Detailed Analysis Results on VMProtect 3.4, *IEEE Access*. 2020.

[41] N. M. Hai, M. Ogawa, Q. T. Tho: Packer Identification Based on Metadata Signature, ACM SSPREW-7, 2017.

[42] P.T.Hung, M.Ogawa: Original Entry Point detection based on graph similarity. *FPS*, LNCS 14551, 2023.

[43] F.Yamaguchi, N.Golde, D.Arp, K.Rieck: Modeling and Discovering Vulnerabilities with Code Property Graphs, *SP* 2014.

[44] Le Vinh: Automatic stub generation from natural language description, Master thesis, JAIST, 2016 September.

[45] H.L.Y.Nguyen: Automatic extraction of x86 formal semantics from its natural language description, Master thesis, JAIST, 2018 March.

[46] Nguyen The Hung: Vulnerabilities detection in binary code, Master thesis, JAIST, 2024 September.

[47] Z.Liy, D.Zouz, S.Xux, X.Ou, H.Jin, S.Wang, Z.Deng, Y.Zhong: VulDeePecker: A Deep Learning-Based System for Vulnerability Detection, *NDSS* 2018.

[48] D.Guo, S.Ren, S.Lu, Z.Feng, D.Tang, S.Liu, L.Zhou, et.al. GraphCodeBERT: Pre-training code representations with data flow, *ICLR*, 2021.

[49] V.A.Nguyen, D.Q.Nguyen, V.Nguyen, T.Le, Q.H.Tran, D.Phung: ReGVD: Revisiting Graph Neural Networks for Vulnerability Detection, *ICSE-Companion*, 2022.

[50] D.Hin, A.Kan, H.Chen, M.A.Babar: LineVD: Statement-level Vulnerability Detection using Graph Neural Networks, *MSR*, 2022.

[51] W.Wang, T.N.Nguyen, S.Wang, Y.Li, J.Zhang, A.Yadavally: DeepVD: Toward Class-Separation Features for Neural Network Vulnerability Detection, *ICSE*, 2023.

[52] S.Nguyen, T.-T.Nguyen, T.T.Vu, T.-D.Do, K.-T.Ngo, H.D.Vo: Code-centric Learning-based Just-In-Time Vulnerability Detection, *archive*, 2023.

[53] Y.Li, S.Wang, T.N.Nguyen: Vulnerability Detection with Fine-Grained Interpretations, *FSE*, 2021.

[54] M.Fu, C.Tantithamthavorn: LineVul: A transformer-based line-level vulnerability prediction, *MSR*, 2022.

[55] Y.Ding, S.Suneja, Y.Zheng, J.Laredo, A.Morari, G.Kaiser, B.Ray: Velvet: a novel ensemble learning approach to automatically locate vulnerable statements, *SANAR*, 2022.

[56] Pham Van Hau, To Trong Nghia, Phan The Duy, A method of generating mutated Windows malware to evade ensemble learning, *Journal of Science and Technology on Information Security*, vol 1, no 18, 2023, pp 47-60. DOI: https://doi.org/10.54654/isj.v1i18.906.

[57] S.Bardin, R.David; J.-Y.Marion: Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes, *SP*, 2017.

[58] Z.Wang, J.Ming, C.Jia, D.Gao: Linear obfuscation to combat symbolic execution, *ESORICS*, LNCS 6879, 2011.

[59] B.Yadegari, S.Debray: Symbolic Execution of Obfuscated Code, *CCS*, 2015.

[60] S.Banescu, C.Collberg, V.Ganesh, Z.Newsham, A.Pretschner: Code Obfuscation Against Symbolic Execution Attacks, *ACSAC*, 2016.

[61] M.Ollivier, S.Bardin, R.Bonichon, J.-Y.Marion: How to Kill Symbolic Deobfuscation for Free (or Unleashing the Potential of Path-Oriented Protections), *ACSAC*, 2019.

[62] M.I.Sharif, A.Lanzi, J.T.Gin, W.Lee: Impeding malware analysis using conditional code obfuscation. *NDSS*, 2008.

[63] M.Sharif, A.Lanzi, J.Giffin, W.Lee: Automatic Reverse Engineering of Malware Emulators, *SP*, 2009.

[64] B.Yadegari, B.Johannesmeyer, B.Whitely, S.Debray: A generic approach to automatic deobfuscation of executable code, *SP*, 2015.

[65] H.Li, Y.Zhan, W.Jianqiang, D.Gu: SymSem: Symbolic Execution with Time Stamps for Deobfuscation, *INSCRYPT*, LNCS 12020, 2019.

[66] M.Liang1, Z.Li1, Q.Zeng, Z.Fang: Deobfuscation of Virtualization-Obfuscated Code Through Symbolic Execution and Compilation Optimization, ICICS, 2017.

[67] T.Blazytko, M.Contag, C.Aschermann, T.Holz: Syntia: Synthesizing the Semantics of Obfuscated Code, *USENIX*, 2017.

[68] J.Salwan1, S.Bardin, M.-L.Potet: Symbolic deobfuscation: from virtualized code back to the original? (long version), *DIMVA*, LNCS 10885, 2018.

ABOUT THE AUTHORS

Mizuhito Ogawa
Workplace:
Old Teachers Network, Japan
Email: mizuhito@gmail.com
Education: Doctor of Science, University of Tokyo
Recent research interests: Formal methods, especially on binary code, e.g., x86, ARM and RISC-V. He also interested in pure theoretical research, such as combinatorics, formal languages and computation theory, especially on the decidability.

Tên tác giả: **Mizihito Ogawa**
Cơ quan công tác: Mạng lưới Cựu Giảng viên Đại học Nhật Bản
Email: mizuhito@gmail.com
Quá trình đào tạo: Tiến sĩ Khoa học tại Đại học Tokyo, Nhật Bản
Hướng nghiên cứu hiện nay: Các phương pháp hình thức, đặc biệt trên mã nhị phân, chẳng hạn như x86, ARM và RISC-V. Ông cũng quan tâm đến các nghiên cứu lý thuyết thuần túy, như tổ hợp, ngôn ngữ hình thức và lý thuyết tính toán, đặc biệt là về tính quyết định.