Jan Harm Kuipers (s1240838) - j.h.kuipers@student.utwente.nl
David Stritzl (s1360752) - d.l.stritzl@student.utwente.nl

Static Disassembly of Obfuscated Binaries
- **Kruegel, Robertson, Valeur and Vigna**

There are two main categories of disassembly techniques: static disassembly and dynamic disassembly. Static disassembly involves processing a binary executable and converting the found opcodes to instructions. Dynamic disassembly techniques involve executing a binary program and monitoring the executed instructions. An advantage of dynamic disassembly is that only reachable instructions are part of disassembler output. At the same time, this is also a disadvantage because it cannot be determined if all possible execution traces of a binary program have been monitored and therefore instructions might be missed by the disassembler.

Static disassemblers come in two main flavours: linear sweep disassemblers and recursive disassemblers. Linear sweep disassemblers such as GNUs *objdump* parse a binary instruction in linear order. This kind of disassembler suffers from data embedded in the binary which might be incorrectly interpreted as instructions. Recursive disassemblers follow the control flow of the program by taking into account branch and call instructions. But the control cannot always be determined due to indirect jumps etc., which complicates disassembly.

Obfuscation of binaries is a process that attempts to modify binary executables in such a way that they become hard to disassemble while limiting the impact on performance and size of the binary. Linn and Debray introduced two techniques for confusing disassemblers. The first technique is inserting junk bytes at unreachable destinations in the machine code, such as directly after a jump instruction. This technique mainly affects linear sweep disassemblers, because recursive disassemblers follow control flow instructions such as jumps. The second technique replaces call instructions for normal functions with a call to a branch function. The call to a branch function executes the original function, but modifies the return address of the function to return to a chosen number of addresses after the original call instruction. This way additional junk bytes can be inserted after every call instruction. A recursive disassembler expects the control flow to resume directly after the call instruction, but with use of a branch function this is no longer the case and the disassembler will interpret the junk bytes as valid instructions (the valid instructions start several addresses after the call instruction).

To improve the disassembly of obfuscated binaries, a novel disassembler is created that can disassemble general obfuscation techniques and the techniques described by Linn and Debray more accurately.

The first step of disassembly is the division of the binary into functions for better parallelisation of the processing workload. In order to identify the functions, a straightforward approach is to extract the targets of all of the call instructions. This would, however, require the call instructions to be already identified, which therefore would require the binary to be

disassembled. Another problem is the obfuscation of call targets by replacing all function calls by a call to a single branching function, which then calls the corresponding function in its place. Since all calls now have a single target, this would also prevent the correct identification of functions. Alternatively, a heuristic approach can be used, where the binary is scanned for typical function prologs, where, among other things, stack space is allocated for use inside the function.

Next, a control flow graph (CFG) can be constructed for each function. In a CFG vertices are represented by so called basic blocks, groups of instructions without branches and edges are represented using control transfer instructions. The traditional approach for this uses recursive disassembly, but produces an incomplete result. Instead, for this disassembler, all possible instructions inside the function, including overlapping instructions, are identified. Using this information, all control transfer instructions (CTIs) that can be found are added to a set of candidates, which can be used to create an initial CFG using recursive disassembly. This implementation of a recursive disassembler has two major differences to typical implementations. For one, it does not assume valid instructions after unconditional jumps. Furthermore, the basic blocks that are found by the algorithm may overlap. This is amended by splitting the two conflicting basic blocks into three successive basic blocks, where the middle basic block will have two predecessing CTIs.

Then, in order to remove the conflicting blocks, all first basic blocks of functions and directly following blocks are deemed valid. If a conflicting basic block can be reached from a single parent block, this parent block is assumed invalid and removed. Next, the conflicting nodes with the least predecessors are removed and after that, the nodes with the least successors. Finally, for all leftover conflicts, one of the blocks is removed at random. This can lead to differing disassembly results.

Lastly, the gaps in between the basic blocks are inspected. These gaps can be junk bytes inserted by the obfuscator or valid blocks, which were not identified because of incomplete CFGs. This can be caused by, for instance, not statically analysable branches or call instructions, for which the succeeding instructions are not inspected by default. For each gap, all possible instruction sequences are identified. A specific sequence is valid if either it reaches up to the next block or end with a CTI. From the leftover sequences, using a statistic and heuristics scores, the most probable one will be chosen.

In addition to the approach described above, tool specific techniques can be applied. Specifically, for the obfuscation method described by Linn and Debray, where call instructions are replaced by branch functions to cover up the real call and return targets. The described function only take static input, ie. the calling address, the obfuscation can be circumvented by simulating the instructions of the branch function, which will lead to the real call and return addresses.

Jakstab: A Static Analysis Platform for Binaries

- **Kinder and Veith**

Jakstab is disassembler of a newly specified class called iterative disassemblers. Jakstab combines several passes of control flow analysis with data flow analysis. It uses an intermediate language Semantic Specification Language (SSL), which is architecture agnostic. An accompanying decompiler called Boomerang, disassembles binary code for several architectures, e.g. x86 and SPARC, to SSL. From the generated SSL Jakstab generates a control flow graph (CFG).

Reverse Code Engineering - State of the Art and Countermeasures

- **Willems and Freiling**

After disassembly of a program, an attempt at decompiling can be made. While perfect decompilation is impossible due to the loss of information that occurs during the compilation stage. Semantically equivalent code can be generated from disassembled binaries. An example of a decompilation tool is the *Hex-Rays Decompiler.*

Several dynamic binary analysis classes can be identified, for example using a debugger with either software or hardware break points (BPs), or virtual machine introspection (VMI), where a binary is run inside a virtual machine which provides access to the execution from the outside.

To mitigate main drawback of dynamic analysis, the fact that only executed binary code is analysed, techniques such as multipath execution or dynamic symbolic execution are used. These techniques attempt to execute all execution paths by forking execution at branch points.

Several countermeasures have been developed against binary analysis, which can be divided in two categories: passive and active methods. Passive methods include the use of *branch functions*, indirect branches, overlapping constants and opaque constants. The use of exception handlers can also be used to force execution out of the normal control flow in order to confuse disassemblers. Packing is also a form of passive protection, where the binary code is encrypted and unpacked on runtime with an included unpacking stub. Polymorphic code changes these unpacking stubs when duplicating itself, to prevent automatic detection of the unpacking stub (AV software detected the unpacking stub to recognize the binary as an encrypted version of certain malware). An even more sophisticated approach translates the binary into a custom instruction set for a customized virtual machine which is used to execute the program.

Active countermeasure against binary analysis are plenty and be further subdivided in several categories: anti-debugging, anti-emulation, anti-virtualization and anti-dumping. Anti-virtualization measures include including rarely used instructions which are not correctly virtualized and detecting facilities for guest-to-host communication. Anti-dumping measures include page-by-page encryption of memory pages, in which an encrypted binary is never

fully unpacked but only parts of the binary are decrypted on access so the binary is never completely loaded in the memory unencrypted.

TIE: Principled Reverse Engineering of Types in Binary Programs

- Lee, JongHyup, Thanassis Avgerinos, and David Brumley. "TIE: Principled reverse engineering of types in binary programs." (2011).

Data type reconstruction is the process of reconstructing source code variables and their types from assembly language code. TIE is a system which can be used in both static and dynamic binary analysis, to infer data types from binary code. TIEs first step is to convert the binary code to a binary intermediate language (BIL). The BIL is a rework of the Vine design from the BitBlaze project. In the case of static disassembly, the methodology proposed by Kruegel and Robertson can be used among others. The second step is variable recovery using an algorithm based on Value Set Analysis (VSA) called the DVSA algorithm.

The final step is type reconstruction and consists of several smaller steps: assign type variables, constraint generation and constraint solving. All variables found in the DVSA analysis are initialized on the “any-type” type. TIE defines a type hierarchy where subtypes exists for several higher level types, e.g. a num32 type (32-bit integer) can be subdivided in signed and unsigned variants. The complete type hierarchy can be seen as a lattice, since certain types can be indistinguishable from each other and thus appear on the same level in the type hierarchy. The constraint generation generates constraints for each variable based on operations on each variable, e.g. a variable used in a if-then always evaluates to a boolean. The constraint solving involves generating a map of all possible types for all variables. Each variable is assigned a lower and upper bound type in the type hierarchy and the distance between the upper and lower bound determines the certainty about the type of the variable .

TIE performs better than professional software such as IDA Pro and better than a competing system called REWARDS, which only works on dynamic binary analysis.

Obfuscation of Executable Code to Improve Resistance to Static Disassembly

- Linn, C., & Debray, S. (2003). Obfuscation of executable code to improve resistance to static disassembly. Paper presented at the Proceedings of the ACM Conference on Computer and Communications Security, 290-299.

In order to make static disassembly harder, one can try to thwart the disassembler. The variable length of x86 instructions makes this more difficult, since the disassembly process will “repair” itself when starting disassembly at an offset location and will end up at the correct instruction sequence within a few instructions.

One way of obfuscating a binary is to insert junk bytes at unreachable “candidate” locations in the binary. These junk bytes must be partial instructions to confuse the disassembler and they must be unreachable at runtime. In order to ensure this, junk bytes can be only inserted

after unconditional control transfer instructions. One can try to magnify the confusion by simulating different junk sequences and choose the one with the least self-repair effect.

To thwart linear sweep disassemblers, junk can be just be inserted at the candidate locations mentioned above. The candidate locations should be considered backwards, so the self-repairing effect of the different junk sequences will not interfere. One way to increase number of candidate locations is to invert conditional branches and thus creating new unconditional branches.

Recursive traversal disassemblers make assumptions about certain control transfers such as returns, which proves to be a weakness that can be exploited to confuse them as well. One method to use this weakness is by using a branch function. This branch function replaces regular function calls and calls the real target function itself based on some parameter, to obfuscate to control flow. Also, it alters the return address of the function, so the function will not return to the instruction directly after the call. This creates space for junk bytes to be inserted and confuse the disassembler since it assumes the the instructions after a call are valid. Another method is the use of opaque predicates: conditional branches that are actually unconditional branches at runtime, since their condition always evaluates to the same result. This method can be also used to create fake jump tables, where only one entry is real and the rest of the table can be filled with junk bytes.

Although, these methods slow down the binaries on average by a factor 1.5, the confusion factors for linear sweep disassembler and recursive traversal disassemblers go up to 88% and 47%, respectively. For the widely used disassembler IDA Pro, the confusion factor goes up to 92% in certain cases.

Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware

- Chen, X., Andersen, J., Morley Mao, Z., Bailey, M., & Nazario, J. (2008). Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. Paper presented at the Proceedings of the International Conference on Dependable Systems and Networks, 177-186. doi:10.1109/DSN.2008.4630086

Computer systems can be categorized into production systems and monitoring systems, which are mainly used to analyse malware execution. Attackers want to differentiate between this system to disable certain code paths on monitoring systems to make the analysis of malware binaries more difficult.

Hardware is one of the factors that can be used to identify the environment. For example, one could check for connected device names and compare them to ones frequently used in emulators like VMWare. Another method is to check for specific internal workings of devices such as the handling of invalid opcodes of a CPU, which will be different in an emulated environment. Also, many debuggers and emulators make use of specific drivers for the tool, which can then be detected by name.

The execution environment also often leaves traces when using an emulator or debugger. VMWare, for instance, uses memory connections between guest and host OS which can be detected. The windows API offers the methods IsDebuggerPresent() and CheckRemoteDebugger() to check for debuggers. Differences in execution path can also provide information, like, for example, certain CPU bugs in emulators or differences in application stack caused by debuggers.

Another method is by checking the application environment. Systems can be scanned for certain virtualization and debugging tools and related applications or registry entries on windows. Also, one can check at runtime for certain processes that are running like debuggers or virtualization related tools.

Lastly, differences in the behaviour of the malware can used to identify monitoring tools. Timing can be used to detect a debugger in single-stepping mode or certain virtualized instructions that perform way worse than on a regular CPU.

Of the 6,300 tested malware samples, almost 5% acted differently in a virtual machine. For debuggers, this number goes up to almost 40%. However, more malware could have remote virtualization detection to prevent installation in the first place. Differences in TCP timing can be detected to differentiate between real and virtualized systems. When an attacker has access to the network, he can also check for certain MAC addresses that are frequently used by virtualization tools.

Simulating some of the debugger and emulator behaviours on a production system, like certain drivers or the windows debugging flags, can lead malware into believing it is run on a monitoring system and therefore prevent it from getting malicious. Of the tested malware samples, 25% showed a reduction of malicious behaviour in the simulated monitoring system. Using imitated network behaviour 70% of the connection attempts could be deterred, compared to the 99% when using a debugger. Simulating debugger and emulator behaviours has been shown to cause only minimal overhead.