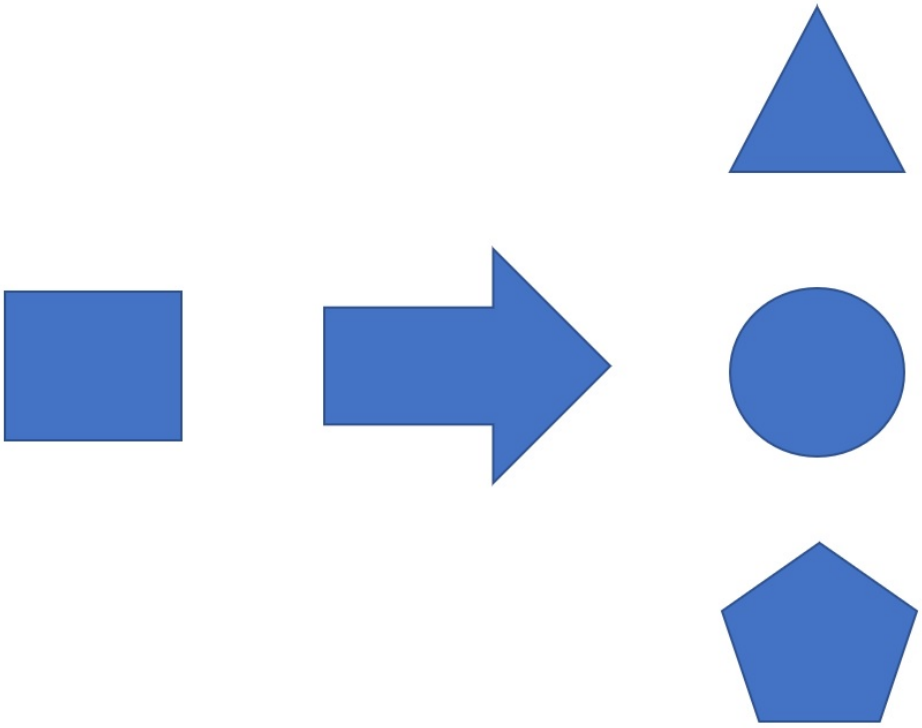

Random Code Variation Compilation

Automated software diversity's performance penalties

Master's thesis in Computer Science – algorithms, languages and logic

Christoffer Hao Andersson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018

Random Code Variation Compilation

Automated software diversity's performance penalties

CHRISTOFFER HAO ANDERSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology
Gothenburg, Sweden 2018

Random Code Variation Compilation
Automated software diversity's performance penalties
CHRISTOFFER HAO ANDERSSON

Supervisor: Thomas Hallgren, Computer Science and Engineering
Examiner: Andreas Abel, Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Division of Computing Science
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: An illustration of a shape that mutates into different shapes to hide its original shape.

Typeset in LaTeX
Gothenburg, Sweden 2018

Random Code Variation Compilation
CHRISTOFFER HAO ANDERSSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

This thesis investigates automated software diversity applied post linking, as a protection method for legitimate applications. A mutation tool was built that can apply different mutation schemes directly on application binaries. Mutation schemes similar to methods used to hide malicious code were implemented. The schemes were applied to a legitimate application. Three properties were then analyzed for each mutation scheme: correctness, uniqueness and performance. Correctness means proper functionality for the mutated code. Uniqueness is how hard it is to recognize the mutated code, knowing the original. Performance is how the mutation affected the application performance-wise. The result showed that the choice of mutation schemes and their parameters greatly affected the uniqueness and performance. It was found that schemes could be grouped into different scheme types, sharing properties in how they work. Finally, it seems like it would be better to apply mutation on a higher level for both performance and uniqueness, either if machine code could be lifted to a higher-level language or if the mutation were integrated directly in the compiler and linker.

Keywords: Automated Software Diversity, Mutation, Compiling, Security, Performance, Metamorphic, Injection, Malicious Code, Virus, Cheat

Acknowledgements

I would first like to thank my Chalmers thesis supervisor, Thomas Hallgren, for his assistance with everything regarding my Master's Thesis. His help has been very valuable for this Master's Thesis.

I also want to thank Meng, my wife, for her continuous support and encouragement. This Master's Thesis would not have been possible without her.

Thank you!

Christoffer Hao Andersson, Gothenburg, January 2018

Contents

List of Figures

1

Introduction

In this section the subject is introduced together with some background. Then possible applications are presented followed by the problem statement and previous research. Finally, some limitations are discussed, and the contribution of this study is presented.

1.1 Overview

Many applications today are vulnerable to attacks based upon signature scans and code injections. Some examples are buffer overflow attacks, viruses, computer game cheats, etc. In this thesis, automated software diversity was applied to legitimate application. By introducing diversity in the binary code, legitimate applications can be protected from malicious code scnas and injections, just as malicious code protects itself against e.g. anti-virus. This can be useful were malicious code depends on already known information about an application, e.g. code signatures or internal data structures.

To measure security and performance penalties, I built a mutation tool. The mutation tool can mutate application binaries directly without need for recompiling, i.e. post-linking. Different mutation schemes were tested and their security and performance penalties were measured. The implemented mutation schemes are similar to software diversity methods used in malicious code. The mutation tool can easily be expanded or modified for further research or applications.

This thesis is aimed towards both an academic audience within the computer science field, as well as any software developer working on applications that might be vulnerable to the security problems described. Very limited compiler support and few tools exists for those techniques described at the time being. This thesis might be of interest for developers facing those security problems.

1.2 Signature scans and code injection

Signature scans can be used to find specific code in an application. It can be seen as a footprint for a certain sequence of assembly code. It is e.g. used in anti-virus software to scan for signatures of known malicious code. Signature scans are also used in malicious code to find code of interest in target applications.

Some examples of signatures:

Listing 1.1: Signature scan example 1

```
8B 41 08          MOV     EAX, DWORD PTR [ECX+8]
85 C0             TEST    EAX, EAX
```

The above Listing 1.1, shows two instruction that looks like ”8B 41 08 85 C0” in flat memory. The simplest signature would then be ”8B 41 08 85 C0”. This signature can be used to search all relevant memory in a process until a hit is found, i.e. the signature is found.

If a signature is too short or too common, multiple hits can be found. Normally, the solution is to add more bytes to the signature if needed, e.g. bytes around the code of interest. However, often just some bytes are needed to uniquely identify a code sequence in a process. We also want to keep signatures as small as possible to not depend on more bytes than necessary.

Listing 1.2: Signature scan example 2

```
8B 45 08          MOV     EAX,                DWORD PTR newSize[EBP]
83 C4 04          ADD     ESP,                4
89 47 04          MOV     DWORD PTR [EDI+4], EAX
8B 0B             MOV     ECX,                DWORD PTR [EBX]
85 C9             TEST    ECX,                ECX
```

When a program is updated and recompiled, constants and static offsets may change, e.g. ”newSize” in Listing 1.2. To prevent that signatures breaks too easy during updates, wildcards can be added to the signature. For Listing 1.2, the following signature can be used ”8B ?? ?? 83 ?? ?? 89 ?? ?? 8B ?? 85”. The signature does not depend on any arguments to the instructions, just the instructions themselves. If the sequence of instructions are found, a hit is reported. Note that it also must be the right version of the instructions, e.g. there exits many OP codes for MOV.

It is also possible to have signatures with an arbitrary number of wildcards. Let a signature independent of previous examples be ”8B ?? 93 [0-5] 89”. Where ”[0-5]” means 0 to 5 wildcards bytes. This can be used to filter out some types of automated software diversity as shown later. Note if signature scans become to general, we starting to get more hits. Normally, just one hit is wanted, to identify the code of interest.

After a signature scan returns 1 hit, the found code can be used. For malicious code, just offsets can be interesting, or code injections can be performed. Offsets can be used to address and access memory. Code injection is basically, overriding the found code with something else. That opens up the possibility to change the functionality of the original application to something else.

Example of some real signatures for CS:GO[1], used in cheats:

```
DumpPatternOffset( "EntityList", "m_dwEntityList", "client.dll",
                   "BB ? ? ? ? 83 FF 01 0F 8C ? ? ? ? 3B F8",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x1, 0x0, ss );

DumpPatternOffset( "WeaponTable", "m_dwWeaponTable", "client.dll",
                   "A1 ? ? ? ? 0F B7 C9 03 C9 8B 44 ? 0C C3",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x1, 0x0, ss );

DumpPatternOffset( "WeaponTable", "m_dwWeaponTableIndex", "client.dll",
                   "66 8B 8E ? ? ? ? E8 ? ? ? ? 05 ? ? ? ? 50",
                   Remote::SignatureType_t::READ, 0x3, 0x0, ss );

DumpPatternOffset( "Extra", "m_dwInput", "client.dll",
                   "B9 ? ? ? ? FF 75 08 E8 ? ? ? ? 8B 06",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x1, 0x0, ss );

DumpPatternOffset( "Extra", "m_dwGlobalVars", "engine.dll",
                   "8B 0D ? ? ? ? 83 C4 04 8B 01 68 ? ? ? ? FF 35",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x12, 0x0, ss );

DumpPatternOffset( "Extra", "m_dwGlowObject", "client.dll",
                   "A1 ? ? ? ? A8 01 75 4E 0F 57 C0",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x58, 0x0, ss );

DumpPatternOffset( "Extra", "m_dwForceJump", "client.dll",
                   "89 15 ? ? ? ? 8B 15 ? ? ? ? F6 C2 03 74 03 83 CE 08",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x2, 0x0, ss );

DumpPatternOffset( "Extra", "m_dwForceAttack", "client.dll"
                   , "89 15 ? ? ? ? 8B 15 ? ? ? ? F6 C2 03 74 03 83 CE 04",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x2, 0x0, ss );

DumpPatternOffset( "Extra", "m_dwForceAttack2", "client.dll"
                   , "89 15 ? ? ? ? 8B 15 ? ? ? ? F6 C2 03 74 06 81 CE ? 20 ? ? A9 ? 20 ? ? BF FD FF FF FF",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x2, 0x0, ss );

DumpPatternOffset( "Extra", "m_dwForceForward", "client.dll"
                   , "8B 15 ? ? ? ? F6 C2 03 74 03 83 CE 08 A8 08 BF FD FF FF FF",
                   Remote::SignatureType_t::READ | Remote::SignatureType_t::SUBTRACT, 0x2, 0x0, ss );
```

Figure 1.1: Screenshot from source code of ”CSGO-Dumper (Y3t1y3t)”

[1]Computer game called ”Counter-Strike: Global Offensive”

1.3 Background

Normal compilers have the emphasis on generating code that will run as efficiently as possible on a specific target. It may be with respect to different parameters like hardware, knowledge of the software, available memory or some other factors. However, with the same inputs (i.e. the same code, platform, version, etc.) the compiler will always generate the same output (i.e. executable). This makes sense performance-wise. Unfortunately, this homogeneity opens for several security problems in some environments and applications.

If the structure of a program is the same for every instance, attackers must only reverse engineer one instance of the software. The gained knowledge can then be used on another instance. The attacks discussed in this thesis are based on signature scans and/or code injections. Those attacks can be found in e.g. buffer overflows, viruses, computer game cheats and other malicious code.

The idea is to introduce a seed to the input of the compiler. For every unique seed, the compiler should generate a unique output, i.e. a unique sequence of machine code instructions. The outputs should still be logically equal. A unique output can be achieved in a lot of different ways, e.g. modifying the instruction scheduling, instruction choices, register choices, the overall structure of the program, calling conventions, control flow, structs/classes member order, virtual table order, etc. [1, 2, 17].

By introducing code variation during the compilation process, the security problems mentioned above can be partially or fully solved. This is because signature scans and code injections are dependent on the machine instructions of the target.

There exist similar concepts for known malicious code e.g. ”Polymorphic code” or ”Metamorphic code” to hide the malicious code from anti-virus software [2, 1, 3]. (”Polymorphic code” is code that e.g. encrypt itself when not executed. ”Metamorphic code” is code that modifies the instructions, but keep the original algorithm intact.) Those techniques are used because anti-virus programs heavily depends on signature scanning for already known signatures [2, 3, 1, 4, 5]. Anti-cheat software basically works the same way [6]. Also in code obfuscation, similar techniques exist where the code is mutated and/or virtualized to make it harder to reverse engineer. However, code obfuscation often comes with performance impacts.

Code variation is introduced to hide signatures and not to alter the function of the application. Code variation, as opposed to code obfuscation, does not need to hide the functionality of the code and, thus, does not come with the same performance penalties. Also, because it is not for virus compilation, the code does not need to be compatible with e.g. a metamorphic engine. The idea is to hide legitimate application signatures from malicious code, just as malicious code hides their signatures from anti-virus software. It is not about making the code hard to reverse engineer. This is a clear difference to code obfuscation, which goal is to make the code as hard as possible to reverse-engineer, not to signature scan.

In performance critical applications such as games, it is of importance to know what sort of performance penalties different software diversity techniques causes. Different techniques may be appropriate for different applications or maybe even different hardware.

By changing signatures of legitimate applications, any malicious code that is dependent on those signatures stops working. For instance, any buffer overflow attack, virus or game cheat dependent on signatures breaks if the signatures changes. Reverse engineering is required again. Even more important, reverse engineering is required every time an application is recompiled with automated software diversity. If updating an application generates a new signature, frequent updates results in frequent reverse engineering for attackers.

Still the anti-virus arms race goes on. Other methods like e.g. jump unrolling [] can be combined with signature scans. However, even if a code sequence can be detected, it is even more complicated to inject code. If applications are distributed (not on a server), the attacker has full control over the execution environment which allows more advanced attacks and analysis.

1.4 Applications

One very rapidly growing market is the gaming market. Naturally, the problem of cheating players is growing as well. Sometimes even legal actions have been the solution [7]. Cheating destroys the experience for other customers and can hurt companies that fails to provide proper protection. One recent example is ”COD: WWII beta” which had cheaters first day on launch [8]. Attackers often earn money from providing cheats for products from other companies [9]. Cheats can essentially be seen as selling extensions without permission.

Normal anti-cheat software heavily relies on signature scans of known cheats [6]. That means new or private cheats are unknown to a signature database at first. By time, they may get added to the database. Still there will be a problem with some cheats around, just as with viruses before they are detected. Cheat uses the same techniques as other malicious code to hide their signatures. The techniques are often some sort of ”Polymorphic code” or ”Metamorphic code” mechanics. This can even make public cheats relatively safe to use. Instead of adding more signatures to the data base, recompiling the game itself with random code variation, breaks all cheats depending on signatures from the game.

Random code variation in combination with frequent updates makes malicious code very cumbersome to maintain, because reverse engineering is required for every update. The need of manual work is likely leading to increased costs for the cheat developers and a more unreliable service for their customers. This can hopefully resulting in less online game cheating. Other benefits are that anti-cheat software can be reduced or completely removed. Removing anti-cheat software can result in better privacy and possibly even better overall performance [10, 11].

Another idea is to let every user have their own unique version of the game. In that way, regular updates can be avoided. However, with a large player base, an update can require millions of compilations. Also, a user can only be semi-trusted and might be willing to use executables provided by a hacker. This approach may work better for server instances, were updates are more infrequent. Each server can have an unique compiled version per licence or instance. That makes buffer overflow attacks very hard if not impossible [12].

1.5 Problem statement

The aim of this master thesis is to test if introducing automated software diversity post-linking, is a practical protection method against signature scans and code injections. This will be tested by applying similar methods as used in malicious code, directly to application binaries. Different mutation schemes or combinations of schemes will be applied, and three properties will be evaluated: correctness, uniqueness and performance.

Theoretically, automated software diversity can be achieved in many ways [17]. However, it might not always be practically feasible due to performance impacts. Another concern is how distinct two mutated outputs are, how unique are they? Uniqueness is important for security, otherwise the uncertainty in the target is lost or not enough. By measuring uniqueness, security of methods can be compared.

1.5.1 The main research question

Is applying automated software diversity post-linking a practical protection method against signature scans and/or code injection for legitimate applications? What properties do mutation have? What are the performance penalties? What is the best configuration in terms of uniqueness contra performance?

This will be answered both by theoretical analysis of different mutation schemes and by implementing a mutator that can apply those mutation schemes on binary files. The modified binaries will be analyzed by both state of the art tools and manual inspection. The performance will be tested by mutating a benchmark application and running performance tests.

1.6 Related work

By introducing diversity in the target, a wide range of attacks can be partly or fully avoided. There exist many ways to diversify software [17]. However, the support in non-commercial and commercial tools is still very limited.

A GCC [2] add-on was recently released that can scramble the order of struct members [13]. It is described as "...obfuscating the internal binary layout of a running kernel, making kernel exploits harder". This is basically one method of automated software diversity that makes it harder for malicious code to use and identify structs. Struct randomization can also be done at run time [18], however this is not currently supported by the GCC add-on.

Another approach is "Random instruction set" [12]. The idea is to have hardware that XOR encrypts all instructions in memory with a random key. When an instruction is fetched for executed, it is decrypted before passed to the processor. A drawback with "Random instruction set" is that it needs an emulator or special hardware [12]. See illustration below in Figure 1.2.

Introducing automated software diversity during the compiling process generates instances that can run on already existing hardware. Just by recompiling the program the same benefits are gained. "Random instruction set" is similar in what they try to achieve but with a different approach. Software diversity at compile time can possibly come with greater performance impacts compared to special hardware. However, it is likely better than an emulator and cheaper than new hardware.

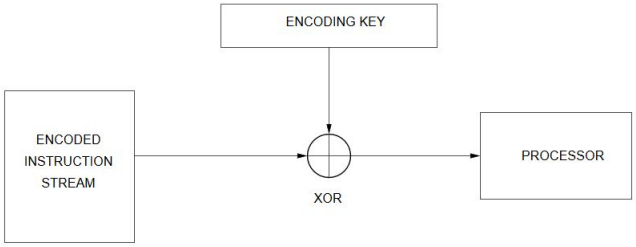

Figure 1: Previously encoded instructions are decoded before being processed by the CPU.

Figure 1.2: Figure taken from [12].

[2] GNU Compiler Collection

1.7 Limitations

Building a compiler and linker from scratch is out of the scope for this thesis, because time and money are limited. However, as seen in [17], post-distribution methods typically have greater overhead than pre-distribution methods. One reason to that pre-distribution methods are better in general is that more information about the application is available, e.g. source code. Pre-distribution methods also provides more flexible code generation, the program does not need to be modified, just generated differently when compiled. When source code is compiled from scratch, no consideration must be taken about already compiled code, since there is none. Those reasons are why it is easier to achieve both better performance and uniqueness with post-distribution methods.

As reported in [20], software diversity often comes with performance overheads. One approach to reduce performance overheads is to use information about sensitive areas like hot paths. A hot path is code that the machine spend much of the execution time at. Cold paths are the opposite, code that the machine spend little to none execution time at. Profile-guided software diversity tries to utilize information about hot paths and cold paths to avoid generating code with high performance overhead in hot paths, while allowing higher performance overhead in cold paths. Normally, hot paths and cold paths are found by profiling the application with expected input. Profile-guided software diversity is a possible improvement for the mutation tool built for this thesis.

For this thesis, code mutation will be performed on already compiled binary files. The binaries must be optimized to get meaningful performance data. The reasons for using mutation on binary files directly, is that it makes testing easier on already existing applications. Modifying binary files directly also avoids the need of recompilation. This is interesting for some applications, e.g. if compile times are long and many instances must be generated. A drawback of modifying binary files directly, is that it is more complex than e.g. modifying object files or assembly code. Code are position or offset dependent so code cannot easily be extended or moved. Branches and sections must therefore be handled very carefully.

In this thesis, some mutation schemes are chosen and discussed. It should be enough to demonstrate the concept of introduction code variation and for getting an idea of performance penalties. However, this should be seen as a general discussion and different applications may respond differently to the same mutation schemes. No prior knowledge is used about applications, like in profile-guided automated software diversity [20]. Also, different hardware may play an important role in performance and as discussed in "Random instruction set"[12] even special hardware is possible.

Testing the concept on a wide range of applications will be out of the scope because of time limitations. The platform will be limited to Windows and Intel's x86 architecture: however, the concept still applies to x64 assembly code as well as any other instruction set available today.

1.8 Contribution

The contribution of this thesis is to give more insight into how automated software diversity affects legitimate applications. It is expected that automated software diversity can solve many existing security problems in million-dollar industries, e.g. the gaming industry. Different mutation schemes were implemented, tested and evaluated, contributing to more knowledge about security and performance penalty for different mutations schemes. Performance penalties are of high importance for most real-world applications. Security contra performance is an interesting aspect that is investigated. Finally, the developed mutation tool used for data collection in this thesis, is provided as an open source tool. Few open source mutation tools exist, so therefore contributing to more open source tools can aid further research or development.

2
Methods

In this section the methods used during this study are presented. Three mutation properties are introduced for comparing different mutation schemes and parameters. The built mutation tool and the implemented schemes are explained. Finally, the test methods used are presented and explained.

2.1 Achieve random code variation

To evaluate random code variation I built a mutation tool. The mutation tool extract arbitrary functions from a binary, apply various mutation schemes, and finally, write the mutated functions back to a new section in the binary. The implemented mutation schemes are inspired by how malicious code hide its code signature [3, 2, 1]. However, any mutation scheme can be implemented.

One main reason to apply mutation post-linking is that mutation can then be applied to already existing binaries without recompilation. Other advantages are that modifying the build chain can be avoided and no source code is needed. This is convenient when the build chain is complicated, which is often the case. To apply mutation post-linking, linking information is used to locate function entries. The mutation tool can also apply mutation to binaries without source or linking information, but reverse engineering is then needed to locate function entries.

2.2 Three properties of mutation

Each mutation scheme will have different properties. In this thesis we look at three properties of each mutation scheme: correctness, uniqueness and performance. Correctness means proper functionality for the mutated code. Uniqueness is how hard it is to recognize the mutated code for an attacker, knowing the original. Performance is how the mutation affects the application performance-wise.

We always want the mutated assembly code to be logically equivalent to the original assembly code. Acceptable uniqueness and performance depends on application requirements. A balance must be found between uniqueness and performance for each application, as explained more below. These three properties are used to measure and compare the quality of different mutation schemes.

In more formal words, let A be an arbitrary assembly code sequence. Define identity, meaning the exact same assembly sequence as $A = A$.

Let

$$M(A, Seed) = A'$$

be a mutation scheme, that takes an assembly code sequence A, a seed $Seed$ and returns a mutated assembly code sequence A'.

Let S be the current state of a machine. Then let

$$E(A, S, X) = S'$$

be an execution function that takes an assembly sequence A, a state S, $X \in \mathbb{N}$ and returns the state S' after A was executed X steps, the sequence returned or the end of the sequence was reached.

2.2.1 Correctness

For any mutation scheme to be useful, correctness must be fulfilled, i.e. a generated assembly sequence A' must be logical equivalent with the input A. Define logical equivalence between two assembly sequences A and A' as

$$A \cong A' \equiv \forall S, X \in \mathbb{N} : E(A, S, X) = E(A', S, X)$$

In this thesis correctness will be tested by comparing output from mutated assembly code with output from the original assembly code. However, software verification or logic proofs can also be used, e.g. like methods used in [21], where a compiler is formally verified.

2.2.2 Uniqueness

For a mutation scheme to be useful, it must not only fulfill the correctness property, but also generate distinct enough output. Otherwise the benefits of automated software diversity are lost. What is distinct enough, depends on the application and the potential attacks.

For an attacker to find or inject code in critical parts of an application, the attacker must first identify where those critical parts are located. Normally, only a small portion of a code sequence is critical. Let $s \in A$ be a signature that exits in A. Then s can be scanned for in A with a signature scan. The simplest decidable case for an attacker is the identity case $A = A'$, i.e. no mutation was performed so if $s \in A$ then $s \in A'$.

By introducing mutation the goal is to achieve $s \notin A'$ when $s \in A$. Let

$$s \in SignatureDB$$

where

$$\forall s \in SignatureDB : s \in A$$

be a set with all or chosen critical signatures for an assembly sequence A.

For this thesis, we define a unique enough mutation scheme M as

$$Unique(M) \equiv \forall Seed, s \in SignatureDB : s \notin M(A, Seed)$$

for a chosen set $SignatureDB$ of signatures. Signatures are determined by application requirements or expected attacks.

Potentially, more advance recognition functions can be used instead of signature scans. Let $s' \in A'$ be the corresponding signature to $s \in A$ after a mutation. To identify $s' \in A'$ knowing $s \in A$, the ideal recognition function for an attacker is $A \cong A'$. If $A \cong A'$ can be identified, then $s \cong s'$ can be identified, because it is a subset of the solution to $A \cong A'$. However, in practice, $A \cong A'$ can not be solved for arbitrary cases as known from the halting problem. Therefore, we expect attackers can identify critical parts somewhere in the domain between the identity function $A = A'$ and the logical equivalence function $A \cong A'$. The uniqueness must outmatch what an attacker can recognize, otherwise an attacker can find critical code and construct attacks.

To measure uniqueness in this thesis, a set of signatures is used. Every generated instance is scanned for all chosen signatures. Fewer signature matches are better. However, in real world examples, a 100% miss rate of signature scans are normally not necessary, it depends on the expected attacks.

2.2.3 Performance

For a mutation scheme M to be useful in practice, it must generate output that have acceptable run-time performance. Let

$$T(A, S, X) = t$$

be a function that returns the time it takes to execute A from state S, X steps, until return or to end of the sequence. Let

$$P(A, M, Seed, S, X) = \frac{T(M(A, Seed), S, X)}{T(A, S, X)}$$

be a function that returns the slowdown factor.

It is expected in general that

$$\forall A, M, Seed, S, X \in \mathbb{N} : T(A, S, X) \leq T(M(A, Seed), S, X)$$

$$\Longleftrightarrow$$

$$\forall A, M, Seed, S, X \in \mathbb{N} : P(A, M, Seed, S, X) \geq 1$$

otherwise an ”optimization” has been performed.

Acceptable performance is defined by application requirements. It may vary greatly between different applications. Some applications may have an upper bound that can not be exceeded. In general, as low average execution time as possible is obviously wanted. To calculate lower and upper bounds for an applications we see M, A, S and X as constants. If M or A are variables, it means the used mutation scheme or application code are changing. If a program always is executed from the start until the end with the same input, S will be a constant state. Let X be the number of steps the program is measured over, or as many steps as the the sequence needs to return. The seed is the only variable, used to generate different instance. The upper and lower bound are then:

Upper bound slowdown factor:

$$\sup_{Seed} P(A, M, Seed, S, X)$$

Lower bound slowdown factor:

$$\inf_{Seed} P(A, M, Seed, S, X)$$

In this thesis performance benchmarks will be used to estimate an average of P.

2.3 Test Methods

For each tested mutation scheme the three properties correctness, uniqueness and performance are be tested.

Correctness is tested by executing mutated versions of a binary and comparing the output with the output from the original binary. Always the same input is used. For the same input the same output is expected, otherwise the binaries are not logically equal which mean $\exists Seed : A \ncong M(A, Seed)$, breaking the correctness property.

Uniqueness is tested by scanning for signatures matching the original binary. Signatures are manually created and must be carefully designed, e.g. offsets, registers and addresses can easily be changed if the binary is recompiled. Signatures should try to avoid dependencies to such factors to be reliable during updates. If a signature $s \in SignatureDB$ exists in $s \in A'$, the uniqueness property is not fulfilled. It is hard to define what enough uniqueness is in general, it depends on the expected attacks.

For the signature scanning itself an external open source tool called YARA will be used. YARA is a well-known tool and used by some of the leading companies when it comes to malware research [14]. YARA has support for searching for custom signature strings, including wildcards and regular expressions. See appendix C for used YARA file.

Performance is measured by running a benchmark application and see how the mutation affects the performance. The performance property is apart from the correctness property hard to say if it is fulfilled. Acceptable performance is defined by application requirements. E.g. in a game the performance requirements can be a certain frame rate, or in a server environment it can be the longest acceptable response time. The mutation must be tested directly on an application to be sure. The performance impact depends on the characteristics of an application, hardware, etc. The goal is to get a general idea of how different mutation schemes affect performance and what causes performance impacts.

A benchmark application is executed to collect data. The same input is always used for all tests. For each tested mutation scheme, 10 different instances are generated. Those instances are executed 10 times each to calculate the average execution time of the instances. The averages of the instances are then used to calculate the average execution time for the mutated application. The average performance penalty can then be compared for each mutation scheme.

The chosen signature scans are tested against each unique instance, i.e. 10 times per mutation scheme. The number of matches are used to describe the efficiency of the tested mutation scheme. Uniqueness and performance can then be compared for each mutation scheme.

2.4 Tested Schemes

Three mutation schemes have been implemented and tested. Each implemented mutation scheme will be described here. The mutation tool will fix changes in offsets so the mutation schemes can operate on assembly code level. E.g. branch and call instructions must be patched when inserting instructions, because it will change jump offsets.

2.4.1 Garbage insertion

Garbage insertion is the simplest mutation scheme. It works by inserting instructions that do not have any meaningful contribution to the application, therefore called garbage. Garbage generation is implemented by pushing a register and popping it immediately afterwards or by just inserting of NOP instructions. More variations of garbage are of course possible to add. The idea is to split up adjacent bytes into two or more separate blocks, to prevent a signature match.

The implemented garbage insertion scheme has three parameters: minimum and maximum instruction block size without garbage instructions, and the maximum number of garbage instructions inserted between the blocks. All block sizes will be randomized but bound by the parameters.

In Listing 2.2, we can see an example of garbage insertion with min and max block size 1. Maximum number of garbage instructions inserted is 5. I.e. 1 to 5 garbage instructions will be inserted between every original instruction.

In Listing 2.3, we can see an example of garbage insertion with min block size 1 and max block size 3. Maximum number of garbage instructions inserted is 5. I.e. 1 to 5 garbage instructions will be inserted between every block of 1 to 3 original instruction.

Examples of garbage insertion:

Listing 2.1: Original code sequence

```
MOV     EDI,    [EBP+0x8]
MOV     ESI,    ECX
NOP     DWORD   [EAX+EAX+0x0]       ; padding added by compiler
MOV     EAX,    EDI
```

Listing 2.2: Garbage insertion with min and max block size 1

```
MOV     EDI,    [EBP+0x8]
NOP
NOP
MOV     ESI,    ECX
NOP
PUSH    EBX
POP     EBX
PUSH    EBX
POP     EBX
NOP     DWORD   [EAX+EAX+0x0]
NOP
NOP
NOP
NOP
MOV     EAX,    EDI
NOP
```

Listing 2.3: Garbage insertion with min block size 1and max block size 3

```
MOV     EDI,    [EBP+0x8]
NOP
PUSH    EAX
POP     EAX
MOV     ESI,    ECX
NOP     DWORD   [EAX+EAX+0x0]
MOV     EAX,    EDI
NOP
NOP
```

To fulfill the correctness property, the state is unchanged of a garbage snippet. This is achieved by choosing garbage snippets that do not affect flags and restore the state if changed before returning.

Garbage insertion will only work against static signature scans. This is because garbage easily can be filtered out through wildcard strings. However, garbage insertion is a very simple technique and serves well as a reference.

Example of how wildcard strings can filter our garbage.

Listing 2.4: Garbage filtering demonstration (Original)

```
8B 7D 08    MOV    EDI, DWORD PTR [EBP+0x8]
89 CE       MOV    ESI, ECX
```

Define a signature for Listing 2.4 as ”8B 7D 08 [0-5] 89 CE”. The signature can handle that 0 to 5 bytes are inserted between the instructions. E.g. both listing 2.5 and listing 2.6 will match.

Listing 2.5: Garbage filtering demonstration (1 bytes inserted)

```
8B 7D 08     MOV     EDI, DWORD PTR [EBP+0x8]
90           NOP
89 CE        MOV     ESI, ECX
```

Listing 2.6: Garbage filtering demonstration (2 bytes inserted)

```
8B 7D 08     MOV     EDI, DWORD PTR [EBP+0x8]
53           PUSH    EBX
5B           POP     EBX
89 CE        MOV     ESI, ECX
```

The expected performance is depending on how frequent garbage is inserted and how much that is inserted. More efficient garbage sequences can be added. E.g. if 5 bytes garbage is needed it is probably more efficient to insert a 5-byte instruction than 5 NOPs if chosen wisely. However, the same instructions can not always be chosen since then it results in a static signature, defeating the purpose of adding garbage from the beginning.

Added padding from the compiler is treated as normal instructions. Inserted garbage will destroy memory alignment created by the compiler. This can potentially decrease performance because frequent jump addresses are often aligned in memory. Implementing support for restoring memory alignment can increase performance if cache misses are a problem.

2.4.2 Scramble

Scramble mutation scheme takes a code sequence and divides it into blocks which are scrambled and reconnected with jumps. The scramble scheme has two parameters: minimum block size and maximum block size. The block size is the number of instructions before a new block is defined and a jump instruction is inserted. The goal is the same as in garbage insertion, to split up adjacent instructions into two or more separate blocks.

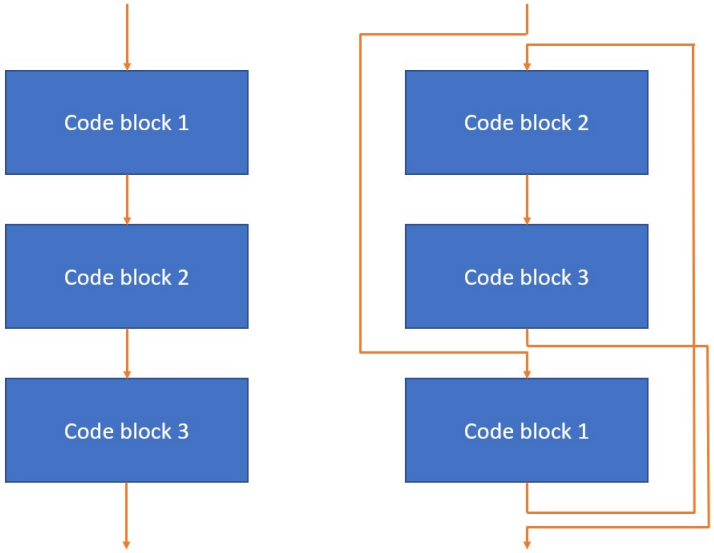

Figure 2.1: Illustration of how the scramble scheme breaks up the execution flow

For real assembly example, see appendix E.

One advantage over garbage insertion is that scramble mutation can protect against wildcard strings, this is because the instructions will no longer be in the original order. However, too large block size (several instructions) leaves signatures intact inside blocks. To signature scan properly, the instructions needs to get unrolled. Unrolling jumps are trivial e.g. if a control-flow graph is built [13]. However, unrolling instructions are more expensive than signature scans, because the instructions must then be disassembled to analyze the code. Signature scans are fast since they only compare bytes.

The correctness property will be fulfilled because unconditional jump instructions do not change the state, apart from the instruction pointer. The original instructions sequence is executed in the original order, apart from inserted jump instructions between the code blocks.

The performance penalty will depend on at least two factors: cache misses and the execution time of jump instructions. Those factors will be affected by the chosen block size as well as memory layout of the blocks. E.g. a small block size will scatter more blocks and also insert more jump instructions. A large block size likely will affect performance less, but as mentioned, with too large blocks, signatures starts to match inside the blocks. Different block sizes have been tested in this thesis to investigate the balance between uniqueness and performance.

The implemented scramble scheme in this thesis does not consider memory alignment or cache misses, performance improvements are therefore possible. E.g. placing connected blocks in the same memory area can reduce cache misses. Alignments for jumps can also be considered, especially were the compiler have already added padding for alignment. E.g. branches, function entry points, etc.

2.4.3 Substitution

Substitution is the last scheme that was tested. The idea with substitution is to break signatures by substitute instructions or instruction sequences. This is the most complicated mutation scheme to implement. Every supported instruction needs special care, especially if flags are modified. The implemented support is limited to PUSH REG, POP REG and MOV REG, REG instructions. The current implementation has predefined substitutions that are generated with help of randomization. Even though not all instructions are supported, the scheme can be used to evaluate if it is worth to implement support for more instructions.

Examples of substitution:

Listing 2.7: Original code sequence

```
PUSH    EBX
```

Listing 2.8: Substituted variant

```
PUSH    EAX
PUSH    EBX
POP     EAX
POP     EBX
PUSH    EAX
PUSH    EAX
PUSH    EBX
POP     EAX
POP     EBX
```

Listing 2.9: Another substituted variant

```
PUSH    EBX
MOV     EBX,    ECX
POP     ECX
PUSH    ECX
PUSH    ECX
MOV     ECX,    EBX
POP     EBX
```

Exact implementations can be found in source code, see appendix A.

The supported instructions do not change any flags. Correctness is fulfilled because, the substituted instructions just performs the same logical operation in another way and do not change any flags. If an instruction modifies the flags, the status of the flags must be restored after the substituted instructions, or special care must be taken. E.g. MOV EAX, EBX is not always replaceable with:

Listing 2.10: MOV instruction replaced by XOR instructions

```
XOR     EAX,    EBX
XOR     EBX,    EAX
XOR     EAX,    EBX
```

since the flags will be different. It depends on if the status of the flags are used afterwards or not. It can be tricky to see if flags are used later or not because of instruction scheduling and pipelining.

Substitution makes it hard to signature scan unless all substitutions are covered. It especially gets tricky when randomization is involved in the substitution. However, all used instructions must be supported to get proper protection.

This post-linking substitution will likely have a significantly higher performance penalty than necessary. This is because instruction are substituted one by one and one instruction is replaced by several other instructions. This blows up the code size relatively fast. A better substitution method is easier to implement at a higher level. If random code variation is supported directly in a compiler more information and options are available.

Example of 3 possibly compatible substitutions:

Listing 2.11: Examples of possible substitutions

```
ADD     EAX,    EAX
MUL     EAX,    2
SHL     EAX,    1
```

The substitutions in Listing 2.11 could possible be interchanged. To safely interchange those instructions, high level information is needed. E.g. if EAX is unsigned, substitutions in Listing 2.11 can be made. However, if EAX is singed, it can not. Also, flags must be treated carefully because different instructions may set flags differently.

Reported in [17], post-distribution methods typically have greater overhead than pre-distribution methods in general. This because less information is available. Another approach to gain more information, is to lift the assembly code to a higher level, e.g. to LLVM or similar. This is not as good as source code, but can easier be analyzed than machine instructions.

2.5 Combining mutation schemes

Different or same mutation schemes can also be combined. Basically, meaning that they are applied one after another. Since the schemes are applied independently after each other, the properties already discussed should still be valid. Otherwise they can not be valid for an original instruction sequence looking like the output from a previous mutation scheme. The hope is to improve the security by gaining advantages that are unique for each combined scheme.

3
Results

In this section the application tested on is presented. The signatures used to test uniqueness are also shown here. Then the test results are presented and analyzed.

3.1 Heap sort benchmark

I implemented a heap sort benchmark to test the mutation schemes. The application is given a text file with numbers to sort. All numbers are loaded into memory before the profiling starts and no outputs are made before the benchmark has ended, hence all IO operations are excluded from the measured benchmark time. The source of the application can be found in appendix A.2.

In total, 24 different mutation scheme configurations were tested with the benchmark application. The original version was tested in the same way and used as a reference point. For each configuration, 10 different instances were generated, i.e. 10 different randomly picked seeds were tested for each configuration.

Each instance was tested against the chosen signatures and the number of matches were counted. One million numbers to sort was given to each instance and the execution time was measured. The execution was repeated 10 times for each instance to measure an average execution time per instance. In total, 2400 executions were performed ($Configurations * Instances * Samples = 24 * 10 * 10$).

Proper function was tested for each instance by checking the min, max, median and total sum, returned by the instance. All executions passed this test.

3.2 Configurations

In the configurations below, a code block is defined a as sequence of instructions with a randomized length. During mutation, the garbage insertion and scramble scheme will split the code into code blocks with varying length until no more code blocks can fit. The length is bound by the min and max block parameters. E.g. with min block 1 and max block 5, code blocks length will randomly vary in the range from 1 to 5 instructions during mutation.

Original:
No modifications.

Garbage (1-1) 1 to 5 garbage instructions:
Insert 1 to 5 garbage instructions between all instructions.

Garbage (1-2) 1 to 5 garbage instructions:
Insert 1 to 5 garbage instructions between every code block. Code block size is randomized dynamically during mutation and will be in the range 1 to 2 instructions.

Garbage (1-3) 1 to 5 garbage instructions:
Insert 1 to 5 garbage instructions between every code block. Code block size is randomized dynamically during mutation and will be in the range 1 to 3 instructions.

Garbage (1-4) 1 to 5 garbage instructions:
Insert 1 to 5 garbage instructions between every code block. Code block size is randomized dynamically during mutation and will be in the range 1 to 4 instructions.

Garbage (1-5) 1 to 5 garbage instructions:
Insert 1 to 5 garbage instructions between every code block. Code block size is randomized dynamically during mutation and will be in the range 1 to 5 instructions.

Garbage (1-1) 1 garbage instructions:
Insert 1 garbage instructions between all instructions.

Garbage (1-3) 1 to 5 garbage instructions x2:
Same as ”Garbage (1-3) 1 to 5 garbage instructions”, but applied twice.

Garbage (1-5) 1 to 5 garbage instructions x2:
Same as ”Garbage (1-5) 1 to 5 garbage instructions”, but applied twice.

Scramble (1-1):
Insert a jump instruction between all instructions.

Scrambe (1-2):
Insert jump instruction between all code blocks. Code block size is randomized dynamically during mutation and will be in the range 1 to 2 instructions.

Scrambe (1-3):
Insert jump instruction between all code blocks. Code block size is randomized dynamically during mutation and will be in the range 1 to 3 instructions.

Scrambe (1-4):
Insert jump instruction between all code blocks. Code block size is randomized dynamically during mutation and will be in the range 1 to 4 instructions.

Scrambe (1-5):
Insert jump instruction between all code blocks. Code block size is randomized dynamically during mutation and will be in the range 1 to 5 instructions.

Scrambe (1-3) x2:
Same as "Scrambe (1-3)", but applied twice.

Scrambe (1-5) x2:
Same as "Scrambe (1-5)", but applied twice.

Garbage(1-3) 1 to 5 + Scramble(1-3):
First apply "Garbage(1-3) 1 to 5", then apply "Scramble(1-3)"

Scramble(1-3) + Garbage(1-3) 1 to 5:
First apply "Scramble(1-3)", then apply "Garbage(1-3) 1 to 5"

Garbage(1-5) 1 to 5 + Scramble(1-5):
First apply "Garbage(1-5) 1 to 5", then apply "Scramble(1-5)"

Scramble(1-5) + Garbage(1-5) 1 to 5:
First apply "Scramble(1-5)", then apply "Garbage(1-5) 1 to 5"

Garbage(1-5) 1 to 5 + Scramble(1-3):
First apply "Garbage(1-5) 1 to 5", then apply "Scramble(1-3)"

Scramble(1-3) + Garbage(1-5) 1 to 5:
First apply "Scramble(1-3)", then apply "Garbage(1-5) 1 to 5"

Garbage(1-3) 1 to 5 + Scramble(1-5):
First apply "Garbage(1-3) 1 to 5", then apply "Scramble(1-5)"

Scramble(1-5) + Garbage(1-3) 1 to 5:
First apply "Scramble(1-5)", then apply "Garbage(1-3) 1 to 5"

3.3 Signatures

Here are the chosen signatures to test against the heap sort benchmark.

Listing 3.1: "Array" signature

```
8b 41 08    mov     eax,    DWORD PTR [ecx+8]
85 c0       test    eax,    eax
```

Listing 3.2: "BubbleUp" signature

```
8b 1c b9    mov     ebx,    DWORD PTR [ecx+edi*4]
8b 14 81    mov     edx,    DWORD PTR [ecx+eax*4]
```

Listing 3.3: "BubbleDown" signature

```
8b ?? ??    mov     ??,     ??
8d ?? ??    lea     ??,     ??
3b ?? ??    cmp     ??,     ??
```

Listing 3.4: "Resize" signature

```
8b ?? ??    mov     ??,     ??
83 ?? ??    add     ??,     ??
89 ?? ??    mov     ??,     ??
8b ??       mov     ??,     ??
85 ??       test    ??,     ??
```

Listing 3.5: "Garbage" signature

```
; exact instruction depends on Mod-R/M
8b ?? 08    mov     ??,     DWORD PTR [??+8]
[0..5] bytes
; exact instruction depends on Mod-R/M
8b ?? b1    mov     ??,     DWORD PTR [ecx+esi*4]
[0..5] bytes
; exact instruction depends on Mod-R/M
8b ?? 93    mov     ??,     DWORD PTR [ebx+edx*4]
[0..5] bytes
89 ?? ??    mov     ??,     ??
```

Listing 3.6: "Substitution" signature

```
8b f2       mov     esi,    edx
8b ?? ??    mov     ??,     ??
```

The two first signatures "Array" signature and "BubbleUp" signature are static byte strings.

The next two signatures "BubbleDown" signature and "Resize" signature are normal wildcard strings. Basically, designed to remove dependence to registers and offsets.

The garbage "Garbage" signature is a special wildcard string that allows some offsets inside the string. Such signatures can very efficiently filter out inserted garbage between the instructions. In this case, up to five bytes can be inserted between the

instructions, the value can also easily be modified if needed to support larger chunks for garbage.

The last tested "Substitution" signature is basically a normal wildcard string as well. The reason for adding it is because it depends on the instruction mov esi, edx which the substitution scheme support. This signature is added to test the substitution scheme.

The YARA signature file can be found in appendix C.

3.4 Test results

The tests were run on a Windows 10 PC, Intel Core i7 3820 @ 2.6GHz with 16 GB ram, for full CPU-Z [1] output see Appendix D. The raw test data table can be found Appendix B.

[1] CPU-Z is a freeware system profiling and monitoring application for Microsoft Windows.

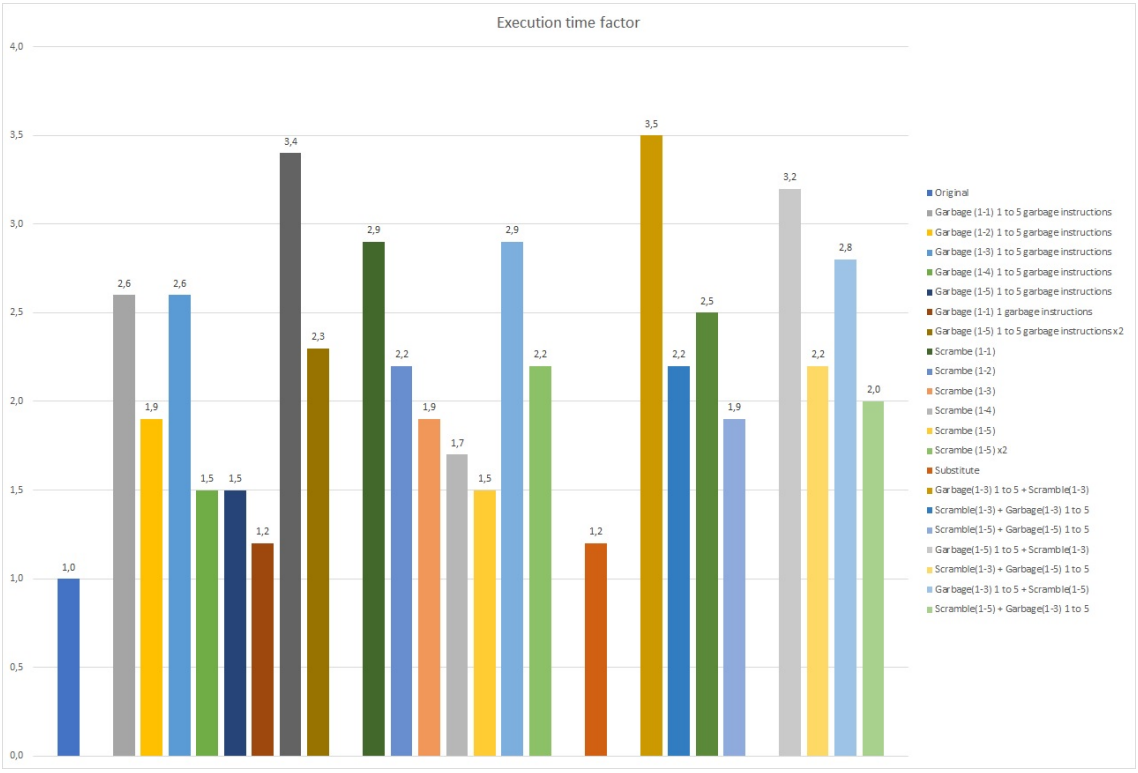

Figure 3.1: Comparison of execution time.

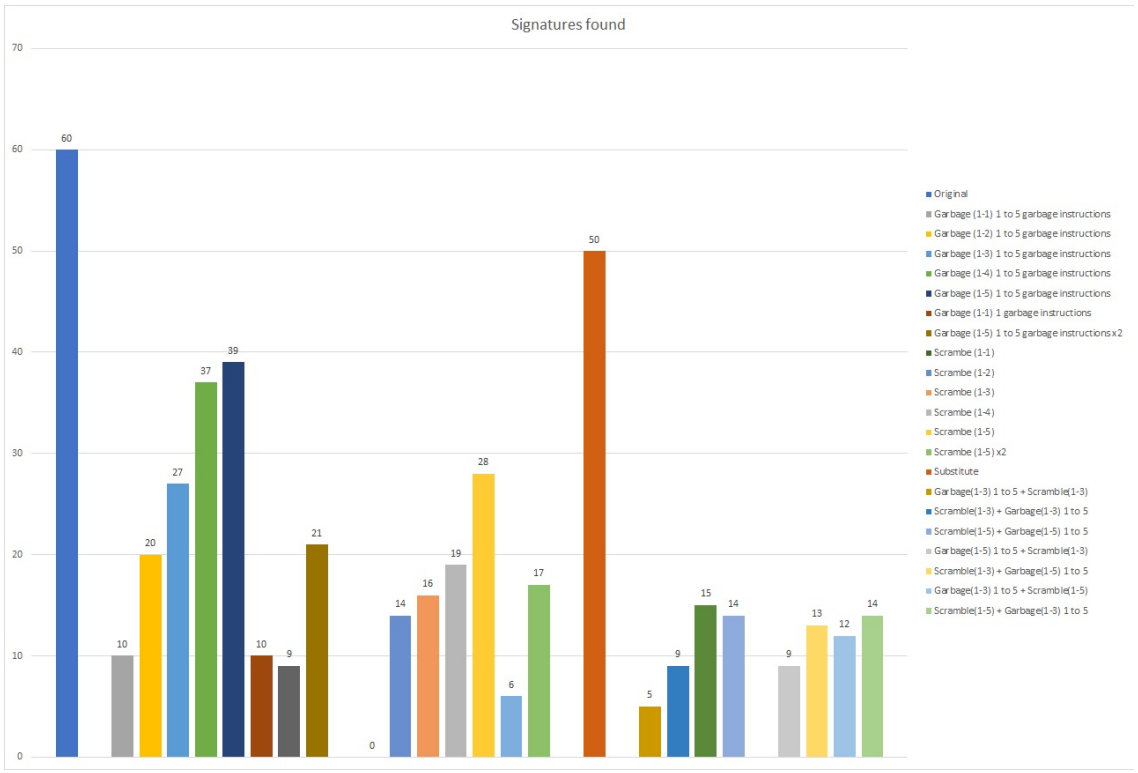

Figure 3.2: Comparison of the numbers of signatures found.

All raw data discussed below can be found in Appendix B. Listing 3.1 visualizes performance penalty factors (less is better). Listing 3.2 visualizes signatures found (less is better). Better uniqueness means less signatures found.

The original application had an average execution time of 130 ms when executed 100 times. Let 130 ms define the execution time factor 1,0. Also, all signatures were obviously found.

As expected, the "Garbage" signature was found for all instances mutated by garbage insertion a single time. When applying garbage insertion twice or combining it with the scramble scheme, the "Garbage" signature was not found for all instances. When combining garbage insertion twice, the garbage sections became too long for the "Garbage" signature. However, the supported length of garbage sections can easily be modified.

The best tested garbage insertion configuration in terms of uniqueness was with min block size 1 and max block size 1, i.e. garbage was inserted between all instructions. If one NOP instruction was inserted instead of up to 5 instructions, the performance was drastically improved. The matching signatures were identical even though less garbage instructions were inserted between the blocks. Worth noticing though is that inserting only one NOP instruction between every instruction generates a static signature for the binaries. However, it shows that since garbage easily can be filtered out in wildcard strings, shorter garbage sections are probably to prefer if preferable at all. Longer garbage sections may not necessarily add more security. Garbage insertion itself does not add enough security to prevent signature scans even if aggressively configured.

The best uniqueness in the test was achieved with the scramble scheme with min block size 1 and max block size 1, i.e. an unconditional branch was inserted between all instructions. The scramble scheme also had better uniqueness than the garbage insertion scheme overall. However, if the "Garbage" signature was excluded, the uniqueness was quite similar to the garbage insertion scheme. Scramble scheme had an overall larger performance penalty compared to the garbage insertion scheme.

For the schemes tested applied twice, the performance penalty was less than two times the single mutation performance penalty. However, the performance penalty contra the number of matched signatures, did never justify choosing the combination. Basically, very similar or better uniqueness can be achieved for similar performance penalty if decreasing the block size instead of applying a scheme twice. The problem with the garbage insertion scheme and scramble scheme, is that the goal is to split an existing signature into different code blocks. When applied twice, instead of decreasing the block size directly, the schemes try to split code inserted from previous pass. Therefore, for these types of schemes, this mostly becomes an inefficient way of decreasing the block size.

When combining the garbage insertion scheme with the scramble scheme, the performance penalty was also less compared to the sum of their single performance penalties. Still it was hard to justify the combinations. Much like applying a scheme twice, there are better options in terms of uniqueness contra performance penalty. The problem is the same as with applying a scheme twice. Since both schemes have the same goal, to break signatures into multiple code blocks. Combining the schemes becomes an inefficient way of decreasing the block size. Applying the scramble scheme last tends to give a little better uniqueness in general. As the results show, the scramble scheme gives better uniqueness in general. Therefore, it has more effect to slightly decrease the scramble max block size instead of the garbage insertion max block size, i.e. it is better to apply the scramble scheme last.

Finally, limited substitution was tested. Since the scheme is very incomplete, it was expected that most signatures matched. However, the "Substitution" signature never matched. That is because it depends on an instruction that was substituted. It is hard to draw any conclusions from the performance data since so many instructions were unsupported.

4
Discussion

In this section the results and the test methods are discussed. Different scheme types and their effect based on the results are discussed. Then some limitations of the testing are discussed. Finally, some further research and the built mutation tool are discussed.

4.1 Results summary

One contribution of this thesis was to investigate whether random code variation is a practical protection method against signature scans and code injection. The scramble scheme was the only scheme in this limited testing that generate a mutation such that 0 signatures were found. The performance impact can not be ignored when the scramble scheme increased the execution time by a factor of 2,9. However, this execution factor is an upper bound which likely can be decreased. With more engineering there are a lot of possibilities and room for improvements.

4.2 Schemes types

Both the garbage insertion scheme and the scramble scheme rely on breaking signatures up into two or more code blocks. The different is that the garbage insertion scheme is breaking signatures by inserting garbage, while the scramble scheme is inserting jumps. The advantage of the scramble scheme in term of uniqueness is that plain signature scans does not work. The code needs to be unrolled, which means the signature scanner must be able to analyze the code and cannot just compare bytes. However, the scrambled blocks cannot be placed right after each other, then the same problem as garbage insertion arises, wildcard strings can be used to filter out garbage or jumps.

As seen in the tests, combining the garbage insertion scheme and the scramble scheme did not have good results in terms of uniqueness per performance penalty. Also, applying the garbage insertion scheme or the scramble scheme twice had very similar problems. Those problems are likely caused by the fact that the garbage insertion scheme and the scramble scheme perform very similar functions. Both schemes rely on breaking up signatures. As we can see from the results, breaking up signatures is best done by picking a small block size, such that signatures cannot fit inside a block. Applying this type of scheme twice only results in an inefficient way of decreasing the block size.

The substitution scheme is another type of scheme. Its goal is not to break up signatures into several different blocks. It is to replace instructions that signatures depend on. One or more operations can likely be done in many ways. E.g. a multiplication of 2 can be replaced with the sum of the value twice or just some bit shifting. Obviously, if the mutation is too static, it is still possible to perform simple signatures scans. However, if performed on a higher level, more options are available. Mutation must not necessarily be performed on single instruction level.

4.3 Limitations

Some things are worth mentioning about the testing. A small single threaded heap-sort benchmarking application was used as performance measurement. It is still a bit uncertain how mutation affects other applications, but some general analysis was made. A single machine was used to run the tests. There was some variation in execution time even though the same input was used. The instances were also executed with the highest priority. The variation probably has to do with the operation system or some other hardware factors.

Multithreading was not tested. For some applications good multithreading may be more important than performance per thread. In such environments, the performance penalty has likely less impact compared to in single threaded applications. However, well threaded applications often tend to be performance critical and may therefore have less margins as well. Schemes that modify the instruction order must also be carefully designed when multithreading is considered. Otherwise e.g. locks may not work properly. Reordering of instructions can be dangerous if not done carefully.

One may think of the idea to only apply mutation to known problem areas, such as some vulnerable functions or classes. That can come with less performance penalty compared to mutating the whole application. It depends on e.g. where the hot paths are. Also, identifying problem areas are not trivial. Attackers can still attack other parts of the application. Some of the original benefits of introduction random code variation are then lost. Knowledge about attacks and what parts that are vulnerable are needed.

Introducing random code variation will not automatically increase security. However, it has great potential for some applications. Games e.g. can be protected from cheats by having regular updates. Servers can be protected by unique instances for each license or server. This make it very hard if not impossible for an attacker to signature scan and inject code. However, just compiling one distribution copy adds very little to no security. One reason it adds some security is that it may not be possible for an attacker to directly compile an exact copy of a target.

Note that unique signatures are not protected against security holes in an API, bugs in software or poor software design but only against signature scans and code injections.

4.4 Mutation properties

There is no standard method of measuring diversification [17]. Three properties were introduces to analyze different mutation schemes. The methods used to analyze those properties in practice can be improved and expanded. E.g. correctness can be verified by logic proofs instead of just testing. A mutated code sequence is only correct when it is logically equivalent to the original code sequence. However, if automated software diversity at higher levels are considered, it might not be possible to compare code sequences directly, because e.g. the structure of the application may differ.

Performance was very hard to forecast. It depends on many factors, algorithms, code layout, input data, underlying systems, hardware, etc. The performance penalties found in this thesis were high, this may be because of the very high uniqueness requirements used. The requirement was that no signature matches were accepted. This requirement may be higher than necessary for most applications. In [19], garbage insertion of NOPs was investigated and the diversification was measured by a percent of found signatures. 26% was suggested as a good ratio of NOP instruction insertions. That means that the probability of inserting a NOP instruction after each instruction was 26%. The found performance penalties were likely better because of the relaxed requirements. However, as the results show, inserting garbage instructions is not a sufficient method against more advanced signature scans. Each application will have different uniqueness and performance requirements.

Depending on what attacks that are expected, there will be different requirements for the mutation to achieve enough uniqueness. For the purpose of this thesis, preventing the used signature scans was defined as enough uniqueness.

The anti-anti-virus arms race will continue. More advanced methods can probably be used, especially for distributed applications, e.g an attacker can better analyze a distributed application, having root access, etc. Similar methods as used in e.g. compiler testing and optimization can potentially be used to optimize a mutated code sequence to some extent. A code sequence does not have to look as the original one, as long as the ”optimized” result is consistent. Reverse engineering and attacks can then be performed on the ”optimized” code sequence instead.

Another option might be to lift the code to a higher-level language like e.g. LLVM. The analysis and modifications can then partly or fully be done in a high-level language. The high-level code can then be compiled back to binary machine code. A function can then be overridden with the new modified code.

4.5 Mutation tool

In [17], the post-distribution methods studied, have higher overheads than the pre-distribution methods. The range reported is from 1% to 250% overhead. Mutating binaries post-linking introduces some complications and limitations. One reason is that less information is available. Mutated functions are also likely to be longer than their original version, the code section must therefore be extended, or a new section must be added. The mutator built in this thesis adds a new section and moves the mutated functions there. This means addresses and offsets needs to be updated, e.g. call and jump instructions. Switch cases must be handled if implemented by a jump table. Mutating binary files directly also involves complications like handling function imports/exports, relocations, etc.

Modifying object files instead of applications simplifies the mutation process to some extent. Relinking binaries and adding an extra section can be completely avoided. However, mutating object files instead of binaries still makes the mutation work on machine instruction level. The main reason this technique was not chosen is that mutation can then not be applied to already compiled applications. The build chain also needs modifications to support modifying object files.

Mutation is likely be easier at a higher level, and as reported in [17], the overheads for pre-distribution methods were lower than for post-distributing methods. Working at machine instruction level limits the mutation options, if the code can not be lifted to a high-level language. However, even if the code was lifted, information is normally lost during compilation.

If mutation is part of a compiler, higher level of mutation is possible, both in the front-end and back-end. Mutation can e.g. be performed in multiple steps during the compilation process, allowing many layers of mutation and utilization of high-level language info. With more information and control over the code, greater software diversification can likely be achieved. Mutation does not have to operate on single machine instruction level, and can e.g. affect the whole structure of the application.

Many mutations are be possible by changing e.g. the overall structure of the program, structs/classes member order, virtual table order, calling conventions, control flow, instruction scheduling, instruction choices, register choices, more sophisticated substitution, etc.

4.6 Mutation tool implementation

The mutator is implemented in roughly 4500 lines C++ code excluding the third-party libraries used. C++ was chosen as the implementation language because of the used third-party libraries. Four third-party libraries were used: JSON (`https://github.com/nlohmann/json`), used for parsing mutation configuration; diStorm3 (`https://github.com/gdabah/distorm`), used to disassembly chosen functions in binary; Asmjit (`https://github.com/asmjit/asmjit`) and Asmtk (`https://github.com/asmjit/asmtk`), used to assemble code back to machine form.

The mutator tool consists of three modules: mutator module, exeedit module and client module. The modules are built in a modular way to easily allow reuse or extension of any module. The source of the mutator can be found in appendix A.1.

4.6.1 Exeedit module

The exeedit module can extract the machine code of arbitrary functions from an executable binary, allowing any modifications of the functions and then writing the modified functions back to a new section. Jump instructions are placed at the old entry point of the functions, pointing to the new entry point in the new section. This is done by parsing the PE header[1] and extracting the chosen functions from the code section. The functions are disassembled by diStorm3 to understand the execution flow. By analyzing the execution flow the functions length can be determined. Multiple return instructions are supported since it sometimes occurs when branches are involved. The machine code plus some code info like length and virtual address are returned. Finally, when the mutated machine code is handed back, a new PE header is generated and the whole executable binary is rewritten.

4.6.2 Mutator module

The mutator module can apply various mutation schemes to any machine code provided. Different random generation algorithms can be used for generating variation. Schemes are applied in passes and the result of one scheme is passed to the next. Schemes are implemented separately and can easily be applied in arbitrary order with any parameters. One or many schemes can be applied during the same execution. The same scheme can even be applied many times with the same or different parameters.

Each scheme starts by disassembling the given machine code with help of diStorm3. Then an intermediate representation is created. The intermediate representation holds information about decoded instructions, e.g. original location, size, etc. This information is later used to recalculate offsets based on new locations. The intermediate representation allows code to be moved or inserted which is not possible directly with machine code. Code is often position or offset dependent, e.g. jumps,

[1] Windows Portable Executable File Format Header

branches and calls. The mutation scheme is then applied on the intermediate representation. Finally, the code is compiled back to machine code, partly by help of Asmjit/Asmtk.

4.6.3 Client module

The client module is a console application which takes a JSON file and a binary file as parameters. The JSON file defines the mutation configuration, i.e. schemes, scheme parameters, order, functions to mutate and random algorithm to use. The client module depends on both the exeedit module and the mutation module and is responsible for executing the given configuration on the chosen binary. The client module can easily be replaced with a GUI based applications instead.

5
Conclusion

To answer the research questions a mutation tool and three mutation schemes were implemented. A test application was mutated by several mutation schemes and with different configurations. The output of each mutation scheme was tested against several signature scans and the performance was measured.

The results showed that different types of mutation schemes and parameters affected the uniqueness and performance greatly. It was not trivial to choose mutation schemes and parameters to get satisfactory results. The tests showed that the garbage insertion scheme did not fulfill the uniqueness property alone. The scramble scheme fulfilled the uniqueness property for all signature scans if aggressive settings were used. However, the performance penalty can not be ignored when it increased the execution time by a factor of 2,9. However, the uniqueness requirements where very high, since no matches were accepted.

Combining schemes with similar function, like the garbage insertion scheme and the scramble scheme, made no sense. The results showed that combining this type of schemes, just led to an inefficient way of decreasing their block size parameter. It was better in terms of both performance and uniqueness, to just decrease the block size directly. Probably, because both schemes works by dividing a signature it into several blocks. Other functioning schemes may still be interesting to combine.

A better implementation is certainly possible to achieve and there are several interesting options to evaluate. Both uniqueness and performance can likely be improved a lot if mutation was applied earlier in the build chain. Its likely that mutation on a higher level, with multiple steps, both add possibilities and simplify the mutation process. This can likely lead to better results. More research is needed to evaluate random code variation further and to push it to commercial compilers.

Bibliography

[1] J. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses" Computer Virology, vol. 4, no. 3, pp. 211-220, 2008.

[2] P. Ször and P. Ferrie, "Hunting for metamorphic" in Virus bulletin conference, Prague, 2001.

[3] P. Szor, "The art of computer virus research and defense" Upper Saddle River: Addison-Wesley, 2005.

[4] "Antivirus.comodo.com" [Online]. Available: https://antivirus.comodo.com/how-antivirus-software-works.php [Accessed 10 February 2016].

[5] "Howtogeek.com" [Online]. Available: http://www.howtogeek.com/125650/htg-explains-how-antivirus-software-works [Accessed 10 February 2016].

[6] "Support.steampowered.com," [Online]. Available: https://support.steampowered.com/kb_article.php?ref=7849-Radz-6869 [Accessed 10 February 2016].

[7] "Polygon" 17 04 2017. [Online]. Available: https://www.polygon.com/2017/4/4/15177818/overwatch-cheat-maker-sued-loses-judgment [Accessed 14 October 2017].

[8] "PC Gamer" 1 10 2017. [Online]. Available: http://www.pcgamer.com/hackers-are-already-infiltrating-the-call-of-duty-wwii-open-beta/comment-jump [Accessed 14 10 2017].

[9] "PC Gamer" 30 04 2014. [Online]. Available: http://www.pcgamer.com/hacks-an-investigation-into-aimbot-dealers-wallhack-users-and-the-million-dollar-business-of-video-game-cheating/ [Accessed 14 10 2017].

[10] C. McSherry, "Electronic Frontier Foundation" 20 October 2005. [Online]. Available: https://www.eff.org/deeplinks/2005/10/new-gaming-feature-spyware [Accessed 10 February 2016].

[11] G. Newell, "reddit" [Online]. Available: https://www.reddit.com/r/gaming/comments/1y70ej/valve_vac_and_trust/ [Accessed 10 February 2016].

[12] S. Gaurav, D. Angelos and P. Vassilis, "Countering Code-Injection Attacks With Instruction-Set Randomization" in CCS'03, Washington DC, 2013.

[13] "Randomizing structure layout [LWN.net]", 11 05 2017. [Online]. Available: https://lwn.net/Articles/722293/ [Accessed 6 9 2017].

[14] B. Dang, A. Gazet and E. Bachaalany, Practical Reverse Engineering, Indianapolis: John Wiley & Sons, Inc., 2014.

[15] "YARA" [Online]. Available: https://virustotal.github.io/yara/ [Accessed 5 9 2017].

[16] "Intel® C++ Compiler in Intel® Parallel Studio XE | Intel® Software" Intel, 2016. [Online]. Available: https://software.intel.com/en-us/c-compilers/ipsxe [Accessed 01 11 2016].

[17] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. Sok: Automated software diversity. In IEEE Symposium on Security and Privacy (Oakland '14), 2014

[18] P. Chen J. Xu Z. Lin D. Xu B. Mao P. Liu "A Practical Approach for Adaptive Data Structure Layout Randomization" *Proceedings of the 20th European Symposium on Research in Computer Security* 2015.

[19] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. Large-scale automated software diversity-program evolution redux. Dependable and Secure Computing, IEEE Transactions on, 2015.

[20] Homescu, A., Neisius, S., Larsen, P., Brunthaler, S., & Franz, M. Profile-guided automated software diversity. In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (pp. 1-11). IEEE Computer Society, 2013

[21] Xavier Leroy. Formal verification of a realistic compiler. Communications of the ACM, 52(7):107–115, 2009.

A

GitHub repositories

A.1 Mutator

https://github.com/farbrorbarbro/ExeMutator

A.2 Test Application

https://github.com/farbrorbarbro/MutateMeDemo

B

Raw test data table

	Average	Execution time factor	Sample 1 avg.	Sample 2 avg.	Sample 3 avg.	Sample 4 avg.	Sample 5 avg.	Sample 6 avg.	Sample 7 avg.	Sample 8 avg.	Sample 9 avg.	Sample 10 avg.	Sig founds	arr_sig	up_sig	down_sig	resize_sig	garbage_sig	substitute
Original	130	1,0	129,68	129,26	128,81	128,60	129,32	128,68	129,25	128,27	128,10	128,58	60	10	10	10	10	10	10
Garbage (1-1) 1 to 5 garbage instructions	340	2,6	357,99	315,25	325,78	306,64	356,98	328,49	398,73	325,25	318,84	365,04	10	0	0	0	0	10	0
Garbage (1-2) 1 to 5 garbage instructions	250	1,9	253,60	266,44	267,40	241,78	238,93	263,17	218,38	231,55	246,07	253,95	20	2	1	0	0	10	7
Garbage (1-3) 1 to 5 garbage instructions	340	2,6	222,06	247,21	231,42	248,86	191,49	256,67	261,40	246,47	228,52	215,95	27	5	6	2	0	10	4
Garbage (1-4) 1 to 5 garbage instructions	200	1,5	210,75	173,54	234,75	177,66	192,41	189,55	192,68	196,56	209,41	189,51	37	7	9	5	0	10	6
Garbage (1-5) 1 to 5 garbage instructions	190	1,5	175,06	215,79	205,10	193,27	190,58	171,14	222,98	208,17	192,83	174,19	39	9	9	4	0	10	7
Garbage (1-1) 1 garbage instructions	160	1,2	162,96	163,22	163,02	162,98	162,96	163,09	162,90	166,62	162,74	164,00	10	0	0	0	0	10	0
Garbage (1-3) 1 to 5 garbage instructions x2	440	3,4	394,88	379,41	591,42	466,46	391,34	530,89	409,51	388,48	349,42	459,15	3	2	1	0	2	1	
Garbage (1-5) 1 to 5 garbage instructions x2	300	2,3	372,91	334,96	262,56	349,61	228,88	263,57	264,40	379,26	289,36	339,71	21	5	6	1	0	4	5
Scramble (1-1)	380	2,9	357,83	385,36	383,71	379,19	385,41	369,74	386,00	370,71	384,73	374,42	0	0	0	0	0	0	0
Scramble (1-2)	290	2,2	307,08	257,40	283,89	296,42	262,07	315,90	297,88	292,49	269,80	290,08	14	6	2	0	0	1	5
Scramble (1-3)	250	1,9	265,03	252,02	249,78	251,54	274,66	244,93	257,71	251,34	232,11	235,32	16	7	6	0	0	0	3
Scramble (1-4)	220	1,7	213,56	241,68	196,44	234,75	246,51	214,88	222,13	200,39	194,43	219,30	19	5	4	4	0	1	5
Scramble (1-5)	200	1,5	189,43	201,47	177,25	214,82	187,23	212,49	204,24	240,02	193,60	194,73	28	8	6	2	1	3	8
Scramble (1-3) x2	380	2,9	366,06	381,16	381,99	360,65	433,28	359,44	364,36	367,81	376,22	369,26	6	4	2	0	0	0	0
Scramble (1-5) x2	290	2,2	326,07	307,97	263,54	239,29	318,34	294,13	311,08	264,45	282,77	275,21	17	7	3	4	0	0	3
Substitute	160	1,2	152,91	162,01	167,12	158,83	166,46	166,23	161,07	160,14	159,61	153,23	50	10	10	10	10	10	0
Garbage(1-3) 1 to 5 + Scramble(1-3)	460	3,5	439,99	447,10	537,71	451,40	476,25	495,44	485,95	417,13	436,31	407,09	5	1	1	0	0	0	3
Scramble(1-3) + Garbage(1-3) 1 to 5	280	2,2	249,14	315,56	295,40	295,69	258,80	273,09	276,41	275,64	261,06	305,14	9	4	4	0	0	0	1
Garbage(1-5) 1 to 5 + Scramble(1-5)	320	2,5	345,14	297,68	336,74	302,68	308,25	303,87	339,08	313,37	310,44	342,28	15	6	4	0	0	1	4
Scramble(1-5) + Garbage(1-5) 1 to 5	250	1,9	232,49	249,55	247,63	253,97	218,29	249,91	259,21	254,80	264,82	231,13	14	4	5	1	0	1	3
Garbage(1-5) 1 to 5 + Scramble(1-3)	410	3,2	426,86	376,99	411,15	440,96	422,28	433,88	358,46	407,56	416,05	397,36	9	3	4	0	0	0	2
Scramble(1-3) + Garbage(1-5) 1 to 5	280	2,2	287,84	253,27	256,64	302,81	270,65	274,38	256,97	276,13	285,90	286,47	13	6	3	0	0	0	4
Garbage(1-3) 1 to 5 + Scramble(1-5)	360	2,8	328,25	363,62	339,28	345,14	339,50	340,46	473,50	366,07	383,70	346,34	12	3	5	1	0	0	3
Scramble(1-5) + Garbage(1-3) 1 to 5	270	2,0	255,67	253,19	270,33	257,59	289,40	282,54	258,61	251,72	342,96	284,28	14	4	3	0	0	4	3

Figure B.1: Data results from testing.

C

YARA signature file

Listing C.1: The used YARA signature file

```
/*
  00000 8b 41 08          mov     eax, DWORD PTR [ecx+8]
  00003 85 c0             test    eax, eax
*/

rule Array_signature
{
        strings:
           $hex_string = { 8b 41 08 85 c0 }

        condition:
           $hex_string
}


/*
  00022 8b 1c b9          mov     ebx, DWORD PTR [ecx+edi*4]
  00025 8b 14 81          mov     edx, DWORD PTR [ecx+eax*4]
*/

rule BubbleUp_signature
{
        strings:
           $hex_string = { 8b 1c b9 8b 14 81 }

        condition:
           $hex_string
}



/*
  0004e 8b 04 99          mov     eax, DWORD PTR [ecx+ebx*4]
  00051 8d 14 99          lea     edx, DWORD PTR [ecx+ebx*4]
  00054 3b 04 b1          cmp     eax, DWORD PTR [ecx+esi*4]
*/

rule BubbleDown_signature
{
        strings:
           $hex_string = { 8b ?? ?? 8d ?? ?? 3b }

        condition:
           $hex_string
}


/*
  00026 8b 45 08          mov     eax, DWORD PTR _newSize$[ebp]
  00029 83 c4 04          add     esp, 4
  0002c 89 47 04          mov     DWORD PTR [edi+4], eax
  0002f 8b 0b             mov     ecx, DWORD PTR [ebx]
  00031 85 c9             test    ecx, ecx
```

```
*/

rule Resize_signature
{
        strings:
           $hex_string = { 8b ?? ?? 83 ?? ?? 89 ?? ?? 8b ?? 85 }

        condition:
           $hex_string
}


/*
  00038 8b 5f 08          mov     ebx, DWORD PTR [edi+8]
  0003b 8b 0c b1          mov     ecx, DWORD PTR [ecx+esi*4]
  0003e 8b 04 93          mov     eax, DWORD PTR [ebx+edx*4]
  00041 89 04 b3          mov     DWORD PTR [ebx+esi*4], eax
*/

rule Garbage_signature
{
        strings:
           $hex_string = {
                8b ?? 08 [0-5] 8b ?? b1 [0-5]
                8b ?? 93 [0-5] 89 ?? ??
            }

        condition:
           $hex_string
}


/*
  00044 8b f2             mov     esi, edx
  00046 8b 47 08          mov     eax, DWORD PTR [edi+8]
*/

rule Substitution_signature
{
        strings:
           $hex_string = { 8b f2 8b }

        condition:
           $hex_string
}
```

D

CPU-Z Output

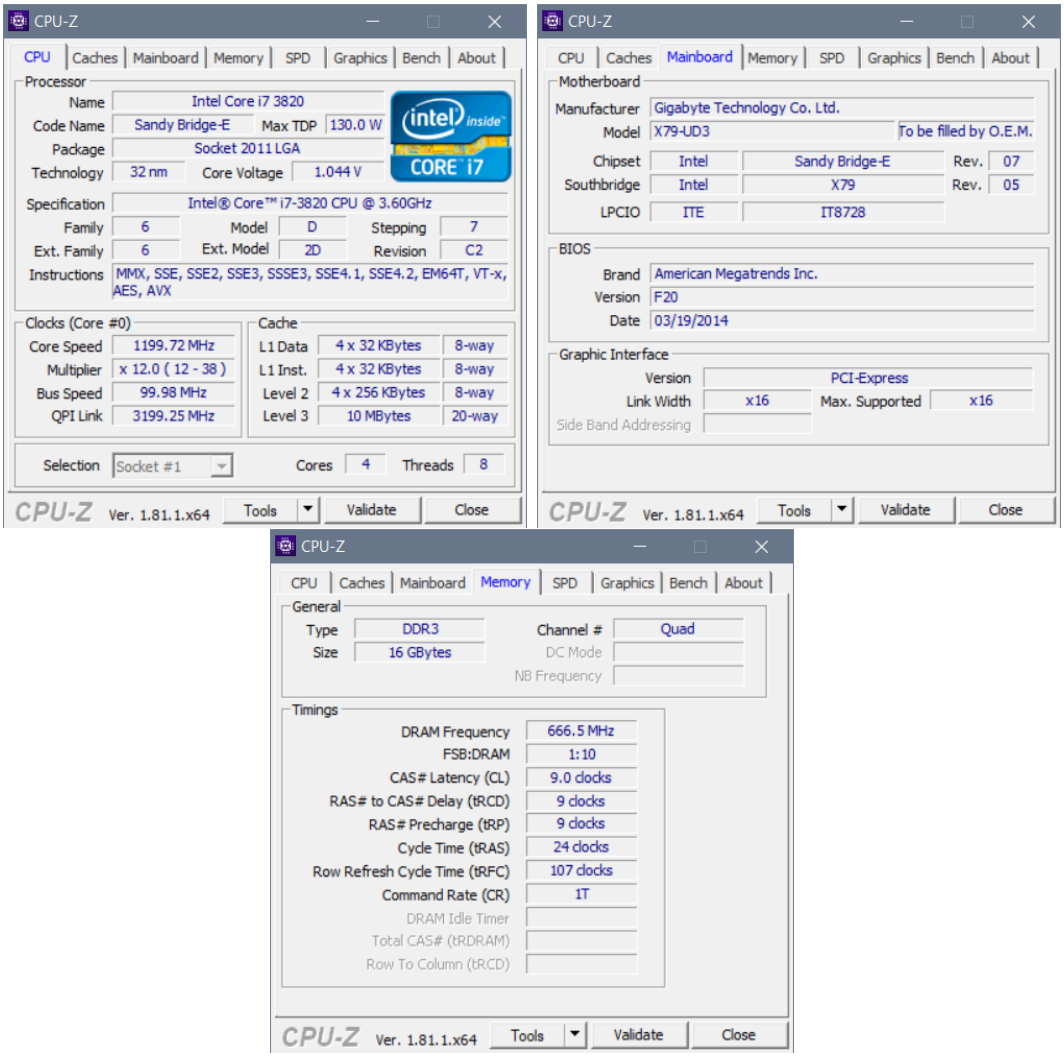

Figure D.1: Output from CPU-Z for the machine tests were executed on.

E

Scramble assembly example

Example of scramble mutation with min block size 1 and max block size 3. I.e. blocks of 1 to 3 instructions will be created and scrambled.

Listing E.1: Original code sequence

```
PUSH    EBP
MOV     EBP,            ESP
PUSH    EBX
PUSH    ESI
PUSH    EDI
MOV     EDI,            [EBP+0x8]
MOV     ESI,            ECX
NOP     DWORD           [EAX+EAX+0x0]       ; padding added by compiler
MOV     EAX,            EDI
CDQ
SUB     EAX,            EDX
SAR     EAX,            0x1
TEST    EAX,            EAX
JLE     0x39
CMP     EAX,            [ESI]
JGE     0x39
MOV     ECX,            [ESI+0x8]
MOV     EBX,            [ECX+EDI*4]
MOV     EDX,            [ECX+EAX*4]
CMP     EBX,            EDX
JGE     0x39
MOV     [ECX+EDI*4],    EDX
MOV     EDI,            EAX
MOV     ECX,            [ESI+0x8]
MOV     [ECX+EAX*4],    EBX
JMP     0x10
POP     EDI
POP     ESI
POP     EBX
POP     EBP
RET     0x4
```

Listing E.2: After scramble mutation.

```
JMP     0x2545

PUSH    EBP
MOV     EBP,            ESP
JMP     0x2595

SAR     EAX,            0x1
JMP     0x25a7

POP     EDI
POP     ESI
JMP     0x2590

MOV     EBX,            [ECX+EDI*4]
MOV     EDX,            [ECX+EAX*4]
CMP     EBX,            EDX
JMP     0x2572

NOP     DWORD           [EAX+EAX+0x0]
JMP     0x259d

JGE     0x2554
MOV     [ECX+EDI*4],    EDX
MOV     EDI,            EAX
JMP     0x25b6

JGE     0x2554
MOV     ECX,            [ESI+0x8]
JMP     0x255b

POP     EBX
POP     EBP
RET     0x4

PUSH    EBX
PUSH    ESI
PUSH    EDI
JMP     0x25c6

MOV     EAX,            EDI
CDQ
SUB     EAX,            EDX
JMP     0x254d

TEST    EAX,            EAX
JLE     0x2554
CMP     EAX,            [ESI]
JMP     0x2582

MOV     ECX,            [ESI+0x8]
MOV     [ECX+EAX*4],    EBX
JMP     0x259d
JMP     0x2554

MOV     EDI,            [EBP+0x8]
MOV     ESI,            ECX
JMP     0x2568
```