Static Analysis on Binary Code

Bin Zeng

Abstract

As the number and sophistication of attacks increase, static analysis gains attention. Since it is binary code that is executed directly on the bare-metal, binary-level static analysis offers root-cause approaches to security problems such as malware detection. In this survey, we start with the challenges to do binary-level static analysis and then transfer to the advantages of carrying out static analysis on binary code. After that, we introduce some typical binary-level static analysis algorithms including disassembly, control flow graph construction, dataflow analysis, alias analysis algorithms et al. Based on the current situation and approaches, we express our own opinions on the future work and propose some preliminary ideas to solve these problems.

1 Introduction

As the number and sophistication of attacks increase, computer security gains more attention and the need to analyze binary code becomes more urgent. Since it is binary code which is executed directly on the bare-metal, binary-level static analysis offers root-cause approaches to security problems. In this survey, we will go through current static analyses on binary code. We assume the target architecture of binary code to be the mainstream architecture x86.

Motivation: Why do we need static analysis on binary code? Static analysis analyzes programs to attain certain properties and behaviors of the code before it is executed. It is widely used in program analysis and compiler optimizations. Static analysis offers the opportunities to determine the properties of programs without running them, thus incurring low or no runtime overhead. Static analysis can be carried out on source code, intermediate representations (IR), assembly code, or even binary code; that is, at any stage before the program is executed. Oftentimes, static analysis is applied to source code and intermediate representations such as Static Single Assignment (SSA) form, as high-level code contains a wealth of structured information. However, binary-level static analysis is gaining popularity due to several trends in software design. The first trend is to extend the functionality of a software by integrating trusted code with untrusted external code. It is extremely difficult or even impossible to implement all the services needed by various customers all at once. For example, almost all the mainstream web browsers support plug-ins or add-ons. Firefox browser relies on Adobe Reader for viewing and printing Adobe Portable Document Format (PDF) files and Adobe Flash Player to deliver Adobe Flash experiences by playing Adobe Flash contents in web pages [30]. Similarly, Chrome browser by Google Inc. supports plug-ins [22]. Another example is the Eclipse development platform [10]. Eclipse provides a core of services for controlling a set of tools working together to support programming tasks. Tool builders contribute to the Eclipse platform by developing Eclipse plug-ins that conform to Eclipse's plug-in contract [10]. Even operating systems load customized policies into kernel for extensibility [8, 28]. The second trend is component-based software development for productivity and cost control [33]. Multiple modules from different vendors are integrated to compose the entire system. Fault isolation among the modules from different vendors is vital for the robustness of the whole system.s In general, binary-level static analysis is needed for these reasons:

1. Source code is unavailable. For malicious code such as viruses, worms, botnets, Trojan horses and untrustworthy extensions such as browser plug-ins, database add-ons and other untrusted code, the source code is either unavailable or untrustworthy. Malware authors usually do not reveal the source code of malwares. Some commercial extensions do not release source code due to the protection of intellectual properties or other considerations. Source-level static analysis cannot be applied without source code. Similarly, IR-level static analysis cannot be carried out easily with only binary code.

Viruses, worms and other malwares simply do not have source code attached. Even if the source code is present, it cannot be trusted since adversaries can simply attach any source code to the binary code and it is generally undecidable or at least not trivial to prove that the binary code is compiled from the attached source code as determining whether two programs are semantically equivalent is undecidable.

2. A compiler might be unfaithful. A compiler might generate incorrect or imprecise code intentionally or unintentionally. Sometimes optimizing compilers optimize code so agressively that the semantics of the binary code might be different from that of the source code. Unfortunately, most languages do not provide constructs to support semantics for security. Thus, compliant compilers do not have the intention or information to preserve what the programmer intends for security. For example, in Fig. 1, an optimizing compiler such as Microsoft Visual C++ .NET which does agressive dead code elimination may optimize away the `memset` function call to improve runtime performance because the definition of password by `memset` is never used afterwards. However, this causes a security vulnerability because the residual value in the password array might be read later by adversaries. The programmer intends to shorten the life time of sensitive information, which is the contents stored in `password` here, by zeroing out the memory location while the compiler optimizes it away to improve runtime efficiency. For this example, programmers can walk around this problem by turning off the optimization performed by the compiler with `volatile` qualifier, command line options, or even writing their own function to wipe out a memory region. However, a typical optimizing compiler does various optimizations and contains a huge code base, thus making a hotbed for bugs. Yang et al. reported 325 bugs in compilers discovered with a randomized test-case generation tool [58]. Some of the bugs cause compilers to silently generate incorrect code when fed with valid input [58]. To accomodate all these cases, programmers have to find out all the bugs in a compiler before they can write secure programs or do translation validation on the generated code. In a nutshell, compilers are too big and complicated to be trusted.

```
int *password = (int *)malloc(sizeof(int)*length); /* to store password    */
read_password(password, length);                    /* read the password    */
process(password, length);                          /* process the password */
memset(password, 0, length);                        /* wipe out the array   */
```

Figure 1: An example demonstrating aggressive optimizations

Similarly, Boehm et al. argue that threads cannot be implemented as a library because without language support compilers are unaware of threads and do agressive optimizations which are totally valid in single-threaded programs but cause multi-threaded programs to fail [9]. This is due to the fact that there is no concept of thread in languages such as C and C++ and compliant compilers are unaware of threads [9].

3. Security rewriting is inappropriate at source level or IR level. Some security strategies analyze and insert security checks into binary code to secure the subject binary code [1, 2, 29, 45]. These approaches need to identify certain machine instructions in the binary code and insert dynamic security checks right before them. At source level or IR level, machine instructions are not even generated. For instance, Control Flow Integrity (CFI, [1, 2]) inserts IDs at branch targets and function starts. Dynamic security checks are placed right before indirect jump, call and return instructions to enforce that execution paths of a program always follow a predetermined control flow graph [1, 2]. The original prototype was implemented as a rewriter at binary level. Similarly, Software Fault Isolation (SFI, [45, 52]) inserts dynamic checking instructions right before each indirect memory visiting instruction to ensure that untrusted code does not visit disallowed data regions [45, 52]. It is not trivial to implement it at source code level or intermediate representation level since there is no concept of indirect memory visiting instructions in C source code or most intermediate representations. In addition, compiler optimizations may invalidate SFI by reordering instructions so that the checking instructions used to sandbox memory operands are not executed effectively before the memory accessing instruction. In addition, source level rewriting puts the whole compiler into the Trusted Computing Base (TCB) and IR-level rewriting has to trust the passes that are executed after the rewriting pass. To put the whole compiler or many passes of a compiler into the TCB impairs the trustworthiness of the security strategies since compilers

are notorious for generating code with security vulnerabilities [4, 58]. By contrast, binary-level static analysis and rewriting do not have to trust a compiler since the final product of a compiler is analyzed and instrumented. Our work is done at binary level thanks to this [59].

4. Some optimizations can be done at binary level to improve runtime performance. Although most program optimizations are done on intermediate representations during compilation, some systems optimize binary code directly with no or limited source code information such as Valgrind [14]. Some optimizations such as peephole optimizations can be applied directly to binary code. Besides, certain global information is generally available at binary level if multiple modules are statically linked together and can enable some optimizations which are impossible in a compiler without whole program optimization due to the needs to support separate compilation. In addition, undefined behaviors of some programming languages such as argument evaluation order in C can be determined on binary code and used by optimizers.

5. Binary code can be reused directly without source code. Sometimes, the source code is lost, unavailable or no longer supported by its original authors and it would be useful to be able to reuse the binary code directly [13]. For example, Trojan horses usually encrypt or compress internet traffic to their clients to hide themselves. If a firewall can extract the decryption and uncompression functions in binary code directly and use them to decrypt and uncompress the traffic, the intrusion would be detected early and thus security would be enhanced.

The reasons to do static analysis on binary code instead of source code or intermediate representations we listed above are far from complete. Static analysis on source code and intermediate representations have their own applications. They are beyond the scope of this survey.

2 Challenges: What makes binary-level static analysis difficult?

In the last section, we talked about the motivations to carry out static analysis on binary code. Static analysis on high-level code is considered to be challenging and binary-level static analysis is even harder. Source code and intermediate representations contain plenty of useful information which can be used by analyzers. The information gets lost or diluted during compilation. For example, in C source code, function signatures contain type information of the functions. A function call can only call to the beginning of a compatible function. In binary code, there are no functions. A jump instruction can jump into the middle of another function, another basic block, or even another instruction. Even an operand of an instruction can be the target of a jump instruction. We list the following reasons why binary-level static analysis is considered to be extremely difficult.

1. Lack of structure information. In general, there is no structure in binary code. The only reliable structure is a byte. Even an instruction is not because two instructions can overlap in x86. Also, there is no explicit control flow graph in binary code. Although for direct branch and call instructions, one can recover the branch targets and call targets easily since their targets are encoded in the instructions themselves, indirect call and jump instructions render it extremely difficult to construct an accurate and complete control flow graph from binary code. Consequently, researchers have to be conservative and make quite approximate assumptions. For example, in the original prototype of CFI [1, 2], an indirect call is assumed to be able to call any function whose address has been taken. In addition, there is no high-level type information in binary code. The only two types in binary code are register and memory location. All high-level type information such as integer, float, and pointers get lost during compilation. Even worse, there is no function or procedure boundary in binary code. One cannot easily tell where a function starts. Usually, symbol tables are absent in binary code or the symbol tables cannot be trusted when they are present since attackers can attach any spurious symbol table to the binary code. It is similar for debugging information. Debugging information is not always present and cannot be trusted either even when it is. It is even impossible to tell code from data when data is intertwined with code in the code section. For example, in Fig. 2, it is impossible to tell statically whether the two while loops will terminate since determining that would solve the halting problem. So, it is impossible to tell whether the assignment between the two while loops will be executed or not.

The function call to `f` is read by the assignment statement. If the first while loop terminates and the second does not, the call to function `f` is data not code since it is only read by the assignment and not executed. If both while loops terminate, the function call to `f` is both data and code since the assignment will read it and it will also be executed.

```
    int a;
    while(){
      ...
    }
    a = *(int *)&&L;
    while(){
      ...
    }
L:
    f();
```

Figure 2: An example showing code intertwined with data

2. Implicit dataflow information. Dataflow information is not explicit in binary code. Binary code does not contain def-use chains, use-def chains or killed set information. Researchers have to expose them through dataflow analysis or other approaches. Value Set Analysis (VSA, [5]) is a typical binary level dataflow analysis algorithm which will be introduced later in this paper.

3. Unmodeled effects. Nowadays, processors are so complicated and expensive to simulate that some features are not completely modeled by compilers such as implicit definitions of the flag register in x86. Many arithmatic and logic instructions in x86 define flags implicitly. For example, `sub` and `shl` instructions define six flags: OF, SF, ZF, AF, PF and CF [23].

```
sub    5, %eax
shl    %ecx, %ebx
jg     label
```

Figure 3: An example illustrating unmodeled effects of the flag register on x86

In the code snippet in Fig. 3: the `sub` instruction subtracts `5` from `eax` and `shl` instruction shifts the value in `ebx` to the left by `ecx` bits (hereafter, assembly code snippet in this survey adopts AT&T syntax). `jg` instruction jumps to `label` if ZF=0 and SF=OF [23]. So which instruction defines the flag register? In actuality, it depends on the content stored in `ecx`. If the content of `ecx` is equal to zero, `shl` instruction has no effect on SF, ZF and PF flags and OF flag is affected only for 1-bit shifts [23]. Otherwise, the flag register is defined by the `sub` instruction.

4. The large number of instructions of x86 and their complexity. x86 is the most prevalent if not popular Instruction Set Architecture (ISA) by far. It is a typical Complex Instruction Set Computing (CISC) architecture. There are hundreds of instructions and the instructions can have variable lengths from one byte to fifteen bytes. The opcodes can be one byte, two bytes or three bytes long. Each instruction can have up to four prefixes, with each taking one byte. A part of an instruction can be a legal instruction. For example, for Linux on x86, the instruction `movl %edx, 0xffffffb8(%ebp)` is encoded using a three-byte sequence `0x89 55 b8` of which `0x55` is also the hexdecimal encoding for instruction `push %ebp`. Jumping to the second byte of the `mov` instruction is one way of executing `push %ebp`. As another example, the hexadecimal encoding for instruction `prefetchnta 0x56e58955` is `0x0f 18 05 55 89 e5 56`. The byte `0x55` is the encoding for instruction `push %ebp` as mentioned earlier. The two-byte sequence `0x89 e5` is the hexadecimal encoding for the instruction `mov %esp, %ebp`. Hexdecimal number `0x56` is the encoding for instruction `push %esi`. By jumping to the fourth byte of the `prefetchnta 0x56e58955` instruction, one can execute the sequence `push %ebp`, `mov %esp, %ebp`

and `push %esi` which can be the prologue of a function. In a nutshell, x86 is very complex and the x86 manuals which describe the semantics and specifications of x86 weigh more than 11 pounds [51].

5. Obfuscation, encryption, compression and packing. Obfuscation is the transformations of programs that preserve the semantics and functionality of the original programs but make reverse-engineering harder. There are several reasons why obfuscation is adopted. In order to protect the intellectual properties such as proprietary algorithms and to prevent licensing verification procedures from being tampered with, commercial softwares are usually obfuscated to prevent reverse-engineering. Besides, malicious code is usually obfuscated to foil detection and analysis [39]. Adversaries can insert effect-free instructions such as `nop`, junk bytes that are never executed into code and reorder the basic blocks to prevent code from being disassembled, analyzed or even decompiled. Also, a branch function [42] can be used to redirect function calls to a single branch function and junk bytes are inserted right after call instructions to deter disassembling [39]. Moser et al. argues that static analysis alone might no longer be sufficient to detect malware due to the fact that opaque constant primitive can be applied in a way that is provably difficult to analyze with a static analyzer [46]. In addition, attackers can just encrypt code statically, decrypt it at runtime and destroy it right after it is executed, which makes static analysis extremely difficult if possible.

6. Self-modifying code. Self-modifying code can modify itself or generate new code during execution. Thus, the binary code in the executable file is not complete. Part of the code is not known until execution. It is not trivial to do static analysis on dynamically generated code since the code itself is not available until runtime. An extreme case is that untrusted code can hide itself by generating malicious code at runtime and destroying it right after it is executed. Since the code that is executed is not available statically, it is impossible or extremely difficult to carry out static analysis on the incomplete code. Here, dynamic analysis can be used as a complement to static analysis. The untrusted code is run on a simulator or virtual machine and its execution trace is generated and fed into a static analyzer such as BitBlaze [51]. After this, static analysis can be done on the execution trace. However, dynamic analysis has a serious weakness, that is, only one execution path for a given input set is generated for a program at a time.

Advantages of binary level analysis. Binary-level static analysis can be a double-edged sword. Although binary-level static analysis is extremely challenging, there are some advantages to do static analysis on binary code. In this paragraph, we will talk about the advantages of binary-level static analysis.

1. Global information is generally present in binary code if multiple modules are linked together into an executable file (those dynamically linked functions are still unavailable till load time or even runtime). Usually binary code is generated by linking separate object files together. Thus, the global information which is hard to obtain during compilation is available in the binary code. In order to support separate compilation, most compilers compile one module at a time (some compilers such as LLVM support whole program optimizations, they are less common and out of the scope of this survey). In general, it is difficult to visit the other modules during compilation. Consequently, at source level and intermediate representation level, static analyzers have to make conservative assumptions about the functions in other modules. For example, in the code snippet in Fig. 4, function `f` resides in file f.c and function `g` is defined in file g.c.

```
int f(int arg){
  if(arg >= 0)
    return 1;
  else return 0;
}
```

f.c

```
extern int f(int a);
int g(int a){
  int b = a * a;
  return f(b);
}
```

g.c

Figure 4: An example demonstrating global information

Function `f` in f.c can only return either 0 or 1. The function `g` in file g.c calls function `f` in f.c. The local variable `b` is always greater than or equal to zero. Thus, the function call `f(b)` in g.c will always

return integer value 1 and thus `g` will always return integer value 1. A compiler without whole program optimization compiles file f.c and g.c separately. When compiling g.c, the compiler has to assume that the call to function `f` in g.c can return any integer value since the information about `f` is unavailable during compilation of g.c. However, in the binary file generated by linking the two modules f.o and g.o together, static analyzers are aware of the definition of both function `f` and `g` and thus can determine that function `g` always returns 1. This example is naively crafted by hand only to show that global information absent during compilation can be present in binary files and useful to enable optimizations which are otherwise impossible.

2. Post-compilation transformations. Some tools instrument binary code after compilation either to optimize binary code or enhance other properties of the binary code such as security [1, 45]. These transformations are applied directly to assembly code or binary code, thus invisible to static analysis at source level or IR level. Malicious code and viruses can even be injected into binary code after compilation. The code injected after compilation is certainly unavailable for source-level static analysis or IR-level static analysis. Binary-level static analysis are aware of all these if they are applied after code injection.

3. Source code unavailability. Due to open source movement, lots of softwares are open source projects. However, proprietary softwares usually do not release their source code because of protection of intellectual properties or other reasons. Sometimes, source code gets lost or is no longer supported by its original authors. Malwares simply do not have source code attached to themselves. Even if source code is present, it cannot be relied upon because there is no proof that the malware is compiled from the attached source code. Without source code, source-level static analysis simply cannot be applied. Binary-level static analysis requires only binary code which is there when it is executed.

4. Multilingual boundary. Unfortunately, there is no panacea programming language yet. There is no programming language good for every task. Some programming languages are better in some aspects than others. The others have different strengths. Consequently, mixed programming or hybrid programming is adopted to take advantage of different features of different languages in large softwares. For example, Java programs can be written in Java and interfaced with C or C++ using Java Native Interface (JNI) to reuse legacy libraries or improve runtime efficiency [34]. In actuality, lots of softwares written in Java use JNI to reuse legacy code written in C [34]. As another example, Fortran programs can also interface with C and C++ code. As a result, static analyzers have to be extended to support multiple languages or be conservative. Multilingual support requires lots of work and is error prone since different languages can have totally different specifications. Binary-level static analyzers can circumvent this problem because it only has to support one language, i.e. the machine language for a certain ISA. No support for high-level source languages is needed.

5. Inlined assembly. Even if the program is written in one high-level language, the source code can still contain inlined assembly code. For example, C++ programs can easily contain some inlined assembly code. Consequently, static analyzers either ignore inlined assembly code or do not extend beyond the inlined assembly code.

6. Binary-level static analysis do not have to worry about optimizations and transformations that are applied to programs during compilation since binary code is the final product of compilation. It is not easy for source-level and IR-level static analysis to predict what compilers and later passes might do during compilation. Binary code is the final product generated in the last phase, the effects of all optimizations are visible to binary-level static analyzers. For example, Software Fault Isolation (SFI) is generally implemented at binary level or assembly level [45, 52].

7. Binary-level static analysis can minimize Trusted Computing Base (TCB). Source-level static analysis need to trust the whole compiler not to do certain transformations that might invalidate the assumptions of static analysis. Thus the whole compiler is in the TCB. Similarly, IR-level static analysis needs to trust the passes which are executed after the analysis pass. However, a compiler is usually too large to be trusted [4, 58]. To the best of our knowledge, there is no compiler which has been proved to always generate binary code faithful to its source code. In other words, compilers are found to generate

binary code which is semantically different from its source code [4, 58]. Binary-level static analysis can minimize the TCB by operating on binary code directly.

3 Current strategies

Static analysis is not a new field. Neither is binary-level static analysis. Ever since the first program was written, static analysis was carried out, not by programs but by programmers themselves. Due to its usefulness and advantages, there is a wealth of publications in the literature on binary-level static analysis and its application to various problems. Due to the large number of publications, completeness is never the goal of this survey paper. We will cover some typical papers in this field. In this section, we will go through the static analyses following the logical order in which they are applied to low-level code. To analyze binary code, the first step is to disassemble the binary code. In the next subsection, we will introduce the state-of-the-art disassemblers used in static analyzers.

3.1 Disassemblers

A disassembler is used to decode the information stored in binary files and translate the machine instructions into an assembly language or an equivalent intermediate representation. In principle, disassembly techniques can be categorized into two domains: dynamic disassembly techniques and static disassembly techniques depending on whether the binary code is executed or not. Dynamic disassemblers execute the binary code on a simulator or emulator such as QEMU [7] and record the execution traces as the binary code is being executed given an input set. With the execution traces, a disassembler can disassemble the instructions that were executed. Since dynamic techniques only record one execution path for a given input set at a time, the execution trace only covers part of the code section. Dynamic disassembly techniques are beyond the scope of this survey. The other approach is static disassembly. A static disassembler takes binary code and tries to disassemble it without running it. In general, there are three static approaches to disassembling binary code. The first approach, called linear sweep, starts at the entry point of binary code and disassembles the instructions in the code section one by one until it reaches the end or illegal instructions (disassemblers can recover from illegal instructions by skipping or other mechanisms). The Unix utility program `objdump` adopts this approach. However, if data is intertwined with code, this approach mistreats data as code and produces incorrect instruction sequences. The second approach, called recursive traversal, disassembles instructions following the control flow graph of the binary code. Since usually there is no predetermined control flow graph attached to binary files, a recursive traversal disassembler has to construct control flow graphs during disassembly. Thus, recursive traversal disassemblers can circumvent data that is embedded in code section. However, this approach also has a drawback, i.e. the control flow graph is not easily constructed when there are indirect jump and call instructions that depend on the contents of registers or memory locations which are statically unknown. The last approach combines linear sweep with recursive traversal. Thus, this approach can have the advantages of both if they are combined appropriately. Kruegel et al. proposed a mixed approach to disassemble obfuscated binary code [39]. We will introduce this approach later in this survey.

There are many popular disassmblers out there. Some of them are widely used and some are not so popular. The most popular disassembler is probably the commercial product IDA Pro widely used by programmers for reverse-enginering [26]. IDA Pro is a recursive traversal diassembler. It takes binary code and outputs the assembly code or equivalent intermediate representation for that binary code following the control flow graph constructed during disassembly. Other than assembly code, IDA Pro can also generate an incomplete control flow graph as a by-product. IDA Pro only generates the control flow edges for direct branch instructions and direct call instructions. Thus, the CFGs generated by IDA Pro do not have the edges induced by indirect jumps and indirect calls. Also, IDA Pro can detect procedure boundaries and calls to library functions using an algorithm, called the Fast Library Identification and Recognition Technology (FLIRT) [31]. IDA Pro has been adopted in many binary code analysis projects such as BitBlaze [51] and VSA [5] et al.

`objdump` is a handy and useful Unix utility tool. It adopts linear sweep technique to disassembly binary code. Binary code is taken in as input and disassembled from the beginning to the end. Data is misinterpreted as code if there is data intertwined in the code. It is not suitable for obfuscated code since it mistreats junk

bytes as code during disassembly. Because of its simplicity as a built-in Unix utility program, it is widely used by programmers for debugging.

Kruegel et al. proposed some techniques to disassemble obfuscated binary code [39]. Their approach mixes linear sweep with recursive traversal to overcome the drawbacks of each and makes use of some statistical methods to diassemble obfuscated code. According to their description, the approach can diassemble a large portion of obfuscated code correctly, especially for the code generated by the obfuscator proposed by Cullen et al. [42]. First, the functions inside the binary code are identified by matching the binary code against function prologues. Usually functions start with prologues that push the frame pointer `ebp` onto the stack and then copy stack pointer `esp` to `ebp` before subtracting stack pointer `esp` to make room for local variables used by the function. The authors of the paper justify this approach by stating that obfuscators generally do not try to hide the function prologues and even if they do, their function identification techniques can be tailored to the corresponding obfuscators. This is a weak argument in my opinion. Adversaries can easily obfuscate a typical prologue with other instructions while preserving the semantics of the original programs. When they obfuscate prologues, they would not tell the disassembler authors to change the prologue identification part. In general, it is weak to assume that obfuscators do not obfuscate some part of a program. Identifying the end of a function is not mentioned in this paper. To my understanding, they assume that functions do not overlap and the start of a function marks the end of the last function. This is also a weakness of this paper. Adversaries can insert any junk bytes at the end of a function and they will be considered to be legal code by their approach. After each function is identified, intraprocedural control flow graphs are constructed by identifying all the direct jump instructions in a function. Since x86 instructions can have variable lengths, each byte address is treated as the beginning of an instruction. Among all the potential instructions, direct jump instructions whose targets are inside the function and conditional branch instructions are selected. These instructions form the set of jump candidates. Then, the jump candidates and the entry point of the function are used as the starting points for a recursive traversal disassembler to construct an initial control flow graph for the function. The initial control flow graph is the supergraph of the real control flow graph. There are some nodes in the initial control flow graph that are not in the real control flow graph. So five steps are taken to resolve the conflicting nodes in the initial control flow graph. The first two steps remove definitely invalid nodes given the assumption that nodes from entry node must be valid and the nodes in conflict with the valid nodes must be invalid and that valid nodes do not overlap. If two nodes in conflict can be reached from a node, i.e. they share an ancestor, the ancestor must be invalid. The other three steps use some heuristics to identify invalid nodes in the initial control flow graph. The third step assumes that a valid node is better integrated into control flow graph. The number of predecessors and direct successors are taken into consideration for conflict resolution. The more predecessors and direct successors, the better integration. The last step removes a node from two conflicting nodes randomly. Obviously the last step can introduce imprecision. After conflict resolvation, there will be no conflict in the control flow graph. However, there might exist some gaps between basic blocks. A simple heuristic is used to disassemble the instructions in gaps assuming that instructions must end at the beginning of the next basic block or with a jump instruction. In addition, some techniques tailored for the obfuscation techniques proposed by Linn et al. [42] were introduced in the paper. The techniques were customized for the branch functions and junk bytes inserted right after unconditional jump instructions and call instructions. Since these are not general techniques, we omit them here.

The approach proposed by Kruegel et al. [39] can effectively disassemble some obfuscated binary code. However, it does not deal with indirect jump instructions or indirect call instructions. Thus the control flow graph created by their disassembler is not complete either. Moser et al. even argued that because opaque constants are provably hard to disassemble, static analysis alone is not enough to detect malware [46]. However, Barak et al. proved in theory that it is difficult to effectively obfuscate programs [6]. Disassembly is never a solved problem. Every time new obfuscation techniques are proposed, corresponding deobfuscation techniques are invented. This will not end any time soon. In the next subsection, we will introduce the state of the art of constructing control flow graphs from binary code.

3.2 Control Flow Graph Construction

Control flow graphs serve as the cornerstone for static analyses. Without a control flow graph, it is impossible or at least very hard to carry out accurate static analysis. Many static analysis algorithms are flow-sensitive

and they require a control flow graph. After binary code is disassembled, the next step is to construct a control flow graph or flesh out the skeleton of a control flow graph if it has been already generated during disassembly. In general, there is no explicit control flow information stored in binary code. Everything needs to be retrieved from the binary code itself. There are several algorithms to construct a control flow graph. For the edges induced by direct branch and call instructions, a disassembler can easily identify their targets and add the edges for them to the control flow graph. For example, IDA Pro constructs a control flow graph with edges induced only by direct branch and call instructions.

The conventional way to construct a control flow graph is to start from the beginning of a function and build the control flow graph while the sequence of instructions are visited. Originally, the control flow graph contains no nodes and no edges. The algorithm starts at the entry point. Every time a jump instruction is identified, the current basic block is terminated. A basic block is a consecutive string of instructions without jump instructions or jump targets in the middle. An instruction in a basic block is always executed before the instructions in later positions in the same basic block and no other instruction is executed in between. For example, in the code snippet in Fig. 5 from the function `get_method` in `gzip.s` from 164.gzip in SPEC CPU2000 compiled by clang with -O3 enabled, the instructions from line 1 to 4 form a basic block, numbered `BB#8`.

```
     ...
1.   movl    insize, %ecx   // BB#8 starts here
2.   movl    inptr, %eax
3.   cmpl    %ecx, %eax
4.   jae     .LBB16_10
5.   leal    1(%eax), %edx  // BB#9 starts here
6.   movl    %edx, inptr
7.   movzbl  inbuf(%eax), %eax
8.   jmp     .LBB16_11
.LBB16_10:                  // BB#10 starts here
9.   movl    $0, (%esp)
10.  call    fill_inbuf
11.  movl    insize, %ecx
12.  movl    inptr, %edx
.LBB16_11:                  // BB#11 starts here
13.  movb    %al, -14(%ebp)
     ...
```

Figure 5: A code snippet from gzip illustrating control flow graph construction

When a conditional jump instuction is visited, the current basic block is connected to two successors. One is the fall-through basic block. The other successor starts with the target of the jump instruction. An unconditional jump only has only one successor which begins with the target of the unconditional jump. For example, in Fig. 5 basic block `BB#8` contains instructions from line 1 to 4. It has two successors `BB#10` and `BB#9`. Basic block `BB#10` contains instructions from line 9 to 12 and `BB#9` contains instructions from line 5 to 8. `BB#9` has only one successor `BB#11` which is omitted in the code snippet. The algorithm keeps running until the whole control flow graph is completed. A call graph is constructed in a similar way. A call graph represents the call-return relationship among functions. The algorithm starts at the entry point and every time a call instruction is encountered, an edge is added to the call graph connecting the caller and the callee.

Although the conventional algorithm is widely used by source level or intermediate representation level control flow analysis, it cannot be borrowed directly by binary level static analysis. Other than direct jump instructions and call instructions which encode their targets in the instructions themselves, there are also indirect jump instructions and indirect call instructions. The targets can come from registers or memory locations. Thus, the targets of indirect jumps and indirect calls cannot always be decoded statically. The targets of indirect jumps and indirect calls can come from global initialized data section such as function tables and jump tables or might depend on the input set which cannot be predicted reliably statically. As

a result, current static analyses are either conservative assuming that each indirect jump can jump to any basic block, any instruction or even the middle of an instruction or optimistic assuming that an indirect jump can jump to only a small set of targets.

A recursive traversal disassmbler disassembles binary code following the control flow graph of the binary code. The algorithm disassembles and constructs control flow graphs at the same time. Disassembly and control flow graph construction facilitate each other. Disassembly can generate assembly code which control flow graph construction algorithm feeds on. In reciprocation, the in-progress control flow graph guides where to disassemble code. Thus, all the disassemblers using recursive traversal generate a control flow graph and a call graph as a side product of disassembly. IDA Pro generates assembly code and a control flow graph for each function and a call graph for the whole program. The algorithm proposed by Kruegel et al. [39] also generates control flow graphs and call graphs. Both of them ignore indirect jumps and calls. In other words, there is no outgoing edges for indirect jumps and calls in the control flow graph and call graph.

Balakrishnan et al. proposed an algorithm called Value Set Analysis (VSA) to analyze the memory contents in binary code statically [5]. Their approach can also detect some of the control flow edges induced by indirect branch and indirect call instructions on the fly. Their goal is to generate the intermediate representation (IR) for binary code that is similar to the IR generated from source code by a compiler. Their prototype takes the assembly code generated by IDA Pro which contains procedure boundaries and incomplete control flow graph and fleshes out the control flow graph during value set analysis. Their approach divides memory space into regions. There is a global region for initialized and uninitialized global data, a region per procedure and a region per heap-allocation statement. Each concrete address can be represented as (memory-region, offset). For example, a local variable `var` in function `f` can be represented as $(AR\_f, -10)$, where `AR_f` is the region for function `f` and -10 is the offset of the address of `var` from the return address. VSA also uses the concept of `a-loc` [5]. An `a-loc` is defined to be the set of locations between two consecutive addresses or offsets [5]. A register is also an abstract store. So is a heap region. Each `a-loc` roughly corresponds to a variable in the source code. Each `a-loc` has an offset which is the offset from the starting address of the `a-loc` to its region. For example, `offset(region, var_10)` is equal to -10, where `var_10` is the `a-loc` for local variable `var`. An abstract store is the mapping from an `a-loc` to its contents. The content of an `a-loc` is represented by a r-tuple of Reduced Interval Congruence (RIC). That is, an `a-loc` is the mapping from $a-loc \rightarrow RIC^r$. VSA is a typical flow-sensitive and context-insensitive worklist algorithm. For each instruction, a transfer function is defined. The algorithm keeps traversing the control flow graph until fixed points are reached. In case the algorithm does not terminate, widening is adopted to ensure termination. The result of this algorithm is the value sets for each `a-loc` at each program location. These value sets can be used to calculate killed, used and possibly-killed sets by reaching-definition analysis. In addition, value sets can also be used to flesh out the skeleton of a control flow graph created by IDA Pro during disassembly. For example, VSA encounters an indirect call instruction `call *0x90480000(, ecx, 4)`. The control flow graph generated by IDA Pro contains no edge for the instruction because it is an indirect call. Let the value set of the index register `ecx` be $[0, 10]$. Then VSA can find the targets of the indirect call instructions in the function table located at `0x90480000` if the function table is in the initialized global data region and the value set of the function table is not changed before the current program point. Due to imprecision and some other restrictions, the targets for some programs cannot be determined statically. VSA generates a report for them and lets the user decide whether the results are acceptable or not.

Some thoughts. To handle indirect jump and indirect call instructions remains an open problem. The current approaches are not satisfactory. An accurate control flow graph serves as the foundation for precise static analysis. Even reliably disassembling a piece of binary code requires a control flow graph. The crucial and hard part of control flow graph construction results from indirect jumps and indirect calls. It might be a good idea to specify the policy that requires the binary code to carry some control flow information for indirect jumps and calls. For example, let the instruction at address `0x80488000` be `call *0x90480000(, eax, 4)`. The targets of the indirect call instructions can be encoded in a special section of the binary code in this form: `0x80488000: 0x80489000, 0x80489010`. The instructions at `0x80489000` and `0x80489010` are the targets of the indirect call instruction. Before the binary code is executed, the control flow information attached to the binary code is verified by a trusted verifier. If the binary code passes the verification, then the control flow information is correct for the program. If not, the binary code is rejected. With the assistance of the control flow information stored within the binary code, all kinds of static analyses can be carried out on the binary code. The binary code can even be decompiled into legitimate C code. Whether the

code is malicious or not can be determined by static analysis with the accurate control flow information. The control flow information can be easily preserved by a compiler and attached to the binary code by a compiler with little or even no help from the programmers, thus incurring no burden on programmers. And program verification techniques can be used to verify whether the control flow information is correct or not. If malicious code does not contain correct control flow information, it would be rejected by the host. If it does, static analysis can be used to detect whether it is malicious or not. Unlike proof-carrying code, it requires little or no intervention from the programmers [47]. Proof-carrying code's scalability problem is also mitigated here because it is comparatively easier to verify a control flow graph than the correctness or safety of a program. The hard part of this idea is to verify the accuracy and correctness of the targets of indirect jumps and calls.

3.3 Dealing with Flag Registers

x86 ISA has a special flag register. Most arithmetic and logic instructions define the flag register [23]. These instructions change the flag register implicitly and their effects on the flag register cannot be disabled and difficult to analyze [51]. The flag register is usually used by branch instructions. As discussed previously, it is not straightforward to model the side effects of instructions on flags in the flag register. Most papers on binary analysis do not mention how they deal with flags. Either they are very conservative or ignore their effects somehow. BitBlaze [51] handles flags in a special way worth discussing.

In order to analyze the side effects of instructions on flags, BitBlaze chooses to expose the effects of each instruction on flags explicitly [51]. For each instruction which defines flags, the corresponding effect of that instruction on the flags is represented by a sequence of Vine instructions [51]. For instance, the `add` instruction in x86 defines CF, PF, AF, ZF, SF and OF flags. Thus, in BitBlaze, for each `add` instruction there is a sequence of Vine instructions which not only model the effects of it on the operands but also on the flags. For example, in Fig. 6 taken from [51], the instruction `addl 2, %eax` is translated into a sequence of Vine instructions. There is a variable for `cf`, `pf`, `af`, `zf`, `sf` and `of` respectively. Each variable is defined by the `add` instruction explicitly. With the effects on flags exposed, dataflow analysis can be carried out on flags more easily.

```
tmp1 = eax;
eax = eax + 2;
// flag register effect exposion
cf = (eax < tmp1);
tmp2 = cast(low, eax, reg8_t);
pf = (!cast(low,
((((tmp2>>7)^(tmp2>>6))^((tmp2>>5)^(tmp2>>4)))^
(((tmp2>>3)^(tmp2>>2))^((tmp2>>1)^tmp2)))), reg1_t);
af = (1==(16&(eax^(tmp1^2))));
zf = (eax==0);
sf = (1==(1&(eax>>31)));
of = (1==(1&(((tmp1^(2^0xffffffff))&(tmp1^eax))>>31)));
```

Figure 6: An example showing explicit flag register effect exposion

3.4 Alias Analysis on Binary Level

This subsection introduces alias analysis on binary level. Alias analysis reasons about whether two memory accesses refer to the same memory location [12, 15, 17, 19, 25, 27, 41, 54]. If two pointers point to the same memory location, they are considered to be aliased. If it cannot be determined whether two pointers point to the same location, they are called may-alias. Alias analysis is also refered to as points-to analysis. The code snippet in Fig. 7 shows a simple example of alias analysis.

There are three possibilities here:

1. The variables `a` and `b` cannot alias;

```
a.mem = -1;
b.mem = 1;
i = a.mem + 1;
```

Figure 7: A simple example showing alias analysis

2. The variables `a` and `b` must alias;

3. It cannot be determined whether `a` and `b` refer to the same memory location.

If `a` and `b` can never alias, then `i = a.mem + 1` can be replaced with `i = 0`. If `a` and `b` always refer to the same memory location, `i = a.mem + 1` can be changed to `i = 2`. In both cases, we are able to perform optimizations with alias information. However, if it cannot be conclusively determined whether a and b alias, the third statement has to be executed to do the computation.

Alias analysis can be classified in terms of flow-sensitivity and context-sensitivity. Flow-sensitive and context-sensitive alias analysis algorithms can improve precision at the cost of complexity and speed. Most alias analyses are performed on intermediate representations before code lowering phases during compilation [32]. Few papers discuss alias analysis on low-level code. Due to its importance for static analysis, alias analysis on low-level code will be introduced next.

Debray et al. introduced a low-level, flow-sensitive, context-insensitive interprocedural pointer alias anlysis algorithm [24]. In this algorithm, addresses are represented by their `mod-k residue`s. That is, only the $m = \log_2 k$ bits of an address is considered. An `address descriptor` is defined be a pair `I` and `M`, where `I` is the defining instruction, `NONE`, or `ANY`, and `M` is a set of `mod-k residues`. So `address descriptor` $A \equiv< I, M >$ denotes a set of `mod-k residue`s relative to the value computed by instruction `I`. The special value `NONE` denotes that the residue values represent residues of absolute addresses and `ANY` indicates the `address descriptor` represents all possible addresses. The authors also defined the effects of individual instructions on the `address descriptor`s of each register. The algorithm only tracks registers. The contents of memory locations are ignored. Also during propagation of `address descriptor`s, the algorithm performs `widening` if the two `address descriptor`s have different defining instructions. The result `address descriptor` will be assigned $\perp$. After the algorithm terminates, two `address descriptor`s are considered to refer to different addresses if they have the same defining instruction and their `residual set`s do not overlap, i.e. the intersection of the two sets equals to $\emptyset$. The algorithm is very inaccurate due to two reasons. The first is that it does not track the contents of memory locations. Once a register is stored into memory and loaded back, the information gets lost. The other reason is that the algorithm uses one abstract address set for each register and performs widening every time the defining instructions are different. This leads to inaccurate results.

Amme et al. proposed a framework to do data dependence analysis on assembly code which can also be used for alias analysis [55]. Their approach models alias analysis as a data flow framework (L, V, F), where `L` is the information set, `V` is the union operator and `F` is the set of semantic functions for the instructions. The content of a register is represented as symbolic values. An initialization point $R_{i,j}$ is a load instruction which defines the register r_i, a call node or an entry node to a function. `init(P)` denotes the set of all initialization points of program `P`. The set of all symbolic values contains all proper polynomial symbolic values and the symbol $\perp$:

$$SV = \{\perp\} \cup \{\sum_{i,j} a_{i,j} \cdot R_{i,j} + c : R_{i,j} \in init(P), a_{i,j}, c \in \mathbb{Z}\}$$

A `k-bounded symbolic value set` is a set with at most `k` sets of polynomial symbolic values.

$$A \in SV_k = \{X : X \subseteq (SV\ \{\perp\}) \bigwedge |X| \leq k\} \cup \{\perp\}.$$

`REGS` denotes the set of all registers. A total map $\alpha : REGS \rightarrow SV_k$ is called a state. When two paths are combined at a joining point, the sets of states are unioned together to form the set for the joining point. If the new set has more than k elements, then the joining point is assigned $\perp$. The semantic functions define

how each instruction update the symbol value sets of each register. The approach is similar to the one proposed by [24] except that the addresses are represented as polynomial symbolic values instead of residual sets.

Every excellent design is a tradeoff between cost and performance. For alias analysis, the cost is the temporal and spatial complexity of the alias analysis algorithm. The performance is the accuracy of the results of alias analysis. These algorithm proposed by Debray et al. [24] is relatively fast but suffers from inaccuracy. The algorithm by Amme et al. is comparatively more precise but slower.

3.5 Application of Static Analysis to Non-security Purposes

Balakrishnan et al. introduced the semantic difference between source code and binary code [4]. They argue that vulnerabilty analysis are better suited at binary level than at source level due to the What You See Is Not What You eXcute (WYSINWYX) phenomenon [4]. They claim that the semantics of binary code is not necessarily equivalent to the source code. For example, in order to improve runtime performance of programs, optimizing compilers eliminate the code used for zeroing out sensitive information during dead-code elimination optimization. They use an example similar to the one in Fig. 1 to illustrate the issue [35].

The Microsoft Visual C++ .NET compiler would optimize away the call to `memset` function completely because the definition by `memset` is not used later and considered to be dead. However, this leads to a security hole since the sensitive information stored at `password` is not destroyed after it is last used and can be read by attackers. The programmer intends to zero out the memory region that stores the sensitive data, password here, while the optimizing compiler optimizes it away for better runtime performance. The semantics of the original program is not preserved during compilation. This paper introduces several examples showing that binary-level analysis is better suited for vulnerability detection. Also, it introduces the platform the authors have developed, called CodeSurfer/x86 [4]. CodeSurfer/x86 is a binary-level analysis framework which requires no source code, no debugging information and no symbol tables. It uses IDA Pro as the front-end to diassemble binary code and build the skeleton of control flow graphs and call graphs. The plug-in to IDA Pro, named Connector [4], is used to do value set analysis. Then, the results of value set analysis is emitted and taken in by CodeSurfer [4], a toolkit for program analysis and inspection. In CodeSurfer, the users can build a collection of IRs including abstract syntax tree, system dependence graph and so on. CodeSurfer/x86 can also do model checking on the binary code [4]. The model checker inside CodeSurfer/x86 uses weighted pushdown system to model possible program behaviors [11, 49]. Users can even output the source code for the binary code if required.

Kruegel et al. [39] introduces an algorithm to construct a control flow graph to facilitate disassembling binary code. Their approach first indentifies each function in binary code by heuristics. The heuristics locates function start addresses by identifying the prologue of each function. On x86 Linux, the prologue of a function is usually `push %ebp` followed by `mov %esp, %ebp` followed by a subtraction instruction which subtracts a constant from `%esp` to make room for the local variables of the function on the stack frame. The authors assume that obfuscators usually do not obfuscate the prologues of functions, thus their approach can identify function start addresses quite accurately. They also use their experimental results to support their assumptions. However, adversaries can easily foil their deobfuscation techniques by replacing the common prologues with other instructions preserving the semantics of the functions. In other words, the assumptions that obfuscators do not obfuscate function prologues are not convincing. Their experiments only test the performance for certain existing obfuscators. They also introduce another approach to identifying function start addresses. The other approach they proposed detects function start addresses by identifying the target addresses of call instructions. Call instructions usually call to function starts. However, this approach can cause circular dependence in my opinion. Function start addresses require the disassembling of call instructions. Disassembling call instructions requires the disassembling of binary code which requires function start addresses.

BitBlaze is a framework for binary code analysis [51]. All kinds of static analysis and dynamic analysis can be done on BitBlaze. BitBlaze has three main parts: Vine, TEMU and Rudder [51]. Vine is responsible majorly for static analysis [51]. TEMU does dynamic analysis and Rudder combines static analysis and dynamic analysis to do concrete and symbolic anlysis [51]. In BitBlaze, binary code is disassembled and then lifted up to Vine Intermediate Language (IL) which is similar to C and assembly language. BitBlaze uses IDA Pro [26], a commercial disassembler, the diassembler proposed by Kruegel et al. [39], and their own linear

sweep based disassmbler to disassemble binary code. After binary code is disassembled, it is transformed into Vine IL [51]. Vine IL explicitly exposes unmodelled side effects such as flag registers. Most of the static analyses are done on Vine IL. Users can even generate C code from Vine IL [51]. Dynamic analysis can be complementary to static analysis. For some binary code such as self-modifying code and encrypted code, static analysis cannot easily handle it because the executed code is not statically available. TEMU is a dynamic analysis component [51]. It is based on QEMU which is a x86 processor emulator. Binary code is first executed on TEMU and the execution trace is generated during execution. Then, the execution trace is fed into Vine to carry out all kinds of static analyses. Rudder is a concrete and symbolic execution component which works as a plug-in to Vine and TEMU [51]. It makes use of the core funcionalities provided by Vine and TEMU to do concrete and symbolic analyses on binary code. It can generate symbolic path predicates that symbolic inputs need to satisfy to to follow that path.

Zhichen Xu et al. introduced safety-checking of machine code [57]. Their approach uses program verification techniques to verify the safety property of machine code directly. The binary code can be generated by any compiler and the source code can be written in any language [57]. As long as nothing unsafe is expressed in the machine code, it is safe to execute the machine code. Their approach has several advantages. The first advantage is that code producers have freedom to choose their favorite languages, even unsafe languages such as C and C++, to write the untrusted code. Code producers are not required to write the extensions in type-safe language such as Java and C#. The second advantage is the decoupling of the safety policies and the source language. Safety policies do not have to be expressed in the source language in which the extensions are written.

Their approach takes four different inputs: untrusted machine code, host-typestate specifications, invocation specification and safety policies [57]. Host-typestate specifications specify the initial state of data before untrusted code is invoked and the pre- and post-conditions for calling host functions and methods. Invocation specification states the initial values passed to the untrusted code when it is invoked by the host. Safety policies include the default safety policies and host-provided policies. Default safety policies include no array out-of-bounds violations, no null-pointer dereferences, no address-alignment violations, no uses of uninitialized variables and no stack-manipulation violations. Host-provided policies specify the accesses permitted on the host data. There are five steps to carry out safety-checking analysis: preparation, typestate propagation, annotation, local verification and global verification. Preparation phase takes host-typestate, invocation specification and safety policies and generate the initial annotations which are composed of the typestates of the input and linear constraints. The second phase, typestate propagation takes the initial annotations and propagates them to each machine instruction with an abstract representation of the memory contents that characterize the state before the execution of that instruction. The third phase local verification checks whether the instructions satisfy the local safety predicates. The last phase global verification uses program verification techniques such as induction-iteration method to verify whether the code satisfies the global safety conditions or not.

Although this approach seems lucrative and viable, its scalability is a serious problem. In the paper, the authors tested their prototype on 13 small toy programs. The largest program is MD5 which contains 883 instructions. There is no evidence in the paper to support that the approach can scale to large programs which contain millions of instructions. The runtime of this approach ranges from 0.06 seconds to 14 seconds on the toy programs. Also, the policies can be hard to encode at the machine code level. When the structure of host data gets complicated, it is hard to write correct safety policies.

4 Static Analysis Applied to Security

This section discusses the application of binary-level static analysis to security problems. Binary-level static analysis has been adopted in malware detection ever since the early days when malwares were invented. Researchers believed that there must be some static differences between benign programs and malicious programs other than their distinct behaviors. Thus, they tried to use static analysis to detect malwares, catch security holes and for other security purposes [3, 16, 18, 20, 36, 43, 44]. For example, virus scanners detect viruses by scanning binaries and match it against the signatures of malwares collected in a database before hand [20]. If there is a match, the scanners report warnings. Otherwise, the binaries are considered benign. Most virus scanners use syntactic signatures to detect malwares. However, syntactic virus scanners

can be thwarted easily through *polymorphism* and *metamorphism* by changing some bits in the malware, thus vulnerable to obfuscation. Some start to use semantic differences to identify malwares [21, 38, 40]. Semantics is extracted from binary code and used to detect malware. Since obfuscation techniques usually change only the syntax of binary code, the semantics still stays the same. In other words, obfuscation techniques usually obfuscate the syntax of binary code but not the semantics. Comparatively, it is harder to change the semantics of programs while still keeping the same functionality. So semantic malware detection techniques are comparatively robust against obfuscation.

We use static analysis to improve the efficiency of current security techniques [59]. In recent years, some ingenious vulnerability mitigations are proposed by researchers to defend against malwares, such as Address Space Layout Randomization (ASLR, [37, 56]), StackGuard and so on. While these vulnerability mitigation techniques can foil lots of attacks, adversaries also have found ways to subvert them [48, 50, 53].

Andreas et al. argue that static analysis alone is not enough for security [46]. They proposed *opaque constants* which are provably difficult for static analysis to analyze precisely binary code. Thus, they argue that static analysis has foundamental limits.

5 Conclusion

In this survey paper, we introduce the motivation, the challenges and the state of art of static analysis on binary code. As the number and seriousness of attacks increase, security becomes more urgent. Binary-level static analysis offers root-cause approaches to security because it is binary code that is executed directly on the bare-metal. However, to carry out static analysis on binary code is extremely challenging because most information available from source code is lost or diluted during compilation. Binary code simply has no type information, no explicit control flow information, no dataflow information, no procedure boundaries and so on. Even part of an instruction can be reused by another one and telling data from code is in general undecidable. In contrast, binary-level static analysis has some advantages versus source-level static analysis. In multiple modules are linked together, global information is available in binary code. There is no need to worry about that compilers might invalidate assumptions. TCB is also minized if static analysis is carried out on binary level. Some static analyses are also investigated here. Disassembling, binary-level CFG construction, VSA and so on are introduced here. This survey is not intended to be complete. The techniques introduced here are just one drop of water in the ocean.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In *ICFEM*, pages 111–124, 2005.

[3] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *In IEEE Symposium on Security and Privacy*, pages 143–159, 2002.

[4] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32:23:1–23:84, August 2010.

[5] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In *CC*, pages 5–23, 2004.

[6] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, 2001.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[8] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.

[9] Hans-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 261–268, New York, NY, USA, 2005. ACM.

[10] Azad Bolour and Bolour Computing. Notes on the eclipse plug-in architecture, July 2003.

[11] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, pages 62–73, New York, NY, USA, 2003. ACM.

[12] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '94, pages 234–250, London, UK, 1995. Springer-Verlag.

[13] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical Report UCB/EECS-2009-133, EECS Department, University of California, Berkeley, Oct 2009.

[14] Filipe Cabecinhas, Nuno P. Lopes, Renato Crisostomo, and Luís Veiga. Optimizing binary code produced by valgrind (project report on virtual execution environments course - avexe). *CoRR*, abs/0810.0372, 2008.

[15] Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. Relevant context inference. In *POPL*, pages 133–146, 1999.

[16] Hao Chen and David A. Wagner. Mops: an infrastructure for examining security properties of software. Technical Report UCB/CSD-02-1197, EECS Department, University of California, Berkeley, Sep 2002.

[17] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM.

[18] Brian Chess. Improving computer security using extended static checking. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 160–, Washington, DC, USA, 2002. IEEE Computer Society.

[19] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 232–245, New York, NY, USA, 1993. ACM.

[20] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

[21] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[22] Google chromium team. Plugin architecture, April 2010.

[23] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, 2009.

[24] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 12–24, New York, NY, USA, 1998. ACM.

[25] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *SIGPLAN Not.*, 29:230–241, June 1994.

[26] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[27] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[28] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[29] Ulfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Ithaca, NY, USA, 2004. AAI3114521.

[30] Alan Grosskurth and Michael W. Godfrey. A case study in architectural analysis: The evolution of the modern web browser. emse, 2007.

[31] Ilfak Guilfanov and DataRescue. Fast library identification and recognition technology, 1997. http://www.hex-rays.com/idapro/flirt.htm.

[32] Bolei Guo, Matthew J. Bridges, Spyridon Triantafyllis, Guilherme Ottoni, Easwaran Raman, and David I. August. Practical and accurate low-level pointer analysis. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 291–302, Washington, DC, USA, 2005. IEEE Computer Society.

[33] George T. Heineman and William T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley Professional, 2001.

[34] Martin Hirzel and Robert Grimm. Jeannie: granting java native interface developers their wishes. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 19–38, New York, NY, USA, 2007. ACM.

[35] Michael Howard. Some bad news and some good news, October 2002. http://msdn.microsoft.com/en-us/library/ms972826.

[36] T. Jensen, D. Le Métayer, and T. Thorn. Verification of control flow based security properties. In *IN PROCEEDINGS OF THE 1999 IEEE SYMPOSIUM ON SECURITY AND PRIVACY*, pages 89–103. IEEE Computer Society, 1999.

[37] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[38] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, Helmut Veith, and Technische Universität München. Detecting malicious code by model checking. In *Proceedings of the 2nd International Conference on Intrusion and Malware Detection and Vulnerability Assessment (DIMVAŠ05), volume 3548 of Lecture Notes in Computer Science*, pages 174–187. Springer Berlin, 2005.

[39] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[40] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.

[41] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[42] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.

[43] Raymond W. Lo, Karl N. Levitt, and Ronald A. Olsson. Mcf: a malicious code filter. *Computers & Security*, 14(6):541–566, 1995.

[44] Michael Dilger Matt Bishop. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131 – 152, 1996.

[45] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[46] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Annual Computer Security Applications Conference*, pages 421–430, 2007.

[47] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[48] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2:20–27, July 2004.

[49] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58:206–263, October 2005.

[50] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[51] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, Newso James, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[52] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[53] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *IN NDSS*, 2003.

[54] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[55] Am Wolfram, Peter Braun, François Thomasset, and Eberhard Zehendner. Data dependence analysis of assembly code. *Int. J. Parallel Program.*, 28:431–467, October 2000.

[56] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *SRDS*, pages 260–, 2003.

[57] Zhichen Xu, Barton P. Miller, and Thomas Reps. Safety checking of machine code. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 70–82, New York, NY, USA, 2000. ACM.

[58] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM.

[59] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *18th ACM Conference on Computer and Communications Security*. ACM, October 2011.