From hack to elaborate technique - A survey on binary rewriting

MATTHIAS WENZL, FH Technikum Wien, Austria
GEORG MERZDOVNIK, SBA Research, Austria
JOHANNA ULLRICH and EDGAR WEIPPL, SBA Research, Austria and CDL-SQI, TU Wien, Austria

Binary rewriting is changing the semantics of a program without having the source code at hand. It is used for diverse purposes such as emulation (e.g., QEMU), optimization (e.g., DynInst), observation (e.g., Valgrind) and hardening (e.g., Control flow integrity enforcement). This survey gives detailed insight into the development and state-of-the-art in binary rewriting by reviewing 67 publications from 1966 up to 2018. Starting from these publications we provide an in-depth investigation of the challenges and respective solutions to accomplish binary rewriting. Based on our findings we establish a thorough categorization of binary rewriting approaches with respect to their use-case, applied analysis technique, code-transformation method and code generation techniques. We contribute a comprehensive mapping between binary rewriting tools, applied techniques and their domain of application. Our findings emphasize that although much work has been done over the last decades, most of the effort was put into improvements aiming at rewriting general purpose applications, but ignoring other challenges like altering throughput-oriented programs, or software with real-time requirements, that are often used in the emerging field of the Internet of Things. To the best of our knowledge, our survey is the first comprehensive overview on the complete binary rewriting process.

CCS Concepts: • **Software and its engineering → Software post-development issues**; *Automated static analysis*; *Dynamic analysis*; • **Security and privacy** → *Software and application security*.

Additional Key Words and Phrases: Binary rewriting, Binary hardening, Static rewriting, Dynamic rewriting , Minimal-invasive, Full-translation, Reassembly

ACM Reference Format:
Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. 2019. From hack to elaborate technique - A survey on binary rewriting. *ACM Comput. Surv.* 52, 3, Article 49 (June 2019), 36 pages. https://doi.org/10.1145/3316415

1 OVERVIEW

"Binary rewriting" describes the alteration of a compiled and possibly (dynamically) linked program without having the source code at hand in such a way that the binary under investigation stays executable [81]. Originally, binary rewriting was motivated by the need to change parts of a program during execution (e.g., run-time patching on the PDP-1 in the 1960's) [92]. Today, binary rewriting has evolved from a hack [92] through a repeatable technique for special purposes like link-time code optimization [5, 62] and performance optimization of win32 programs [112] to a plethora of approaches with applications in multiple domains. Popular applications are:

We express our gratitude to our reviewers who greatly helped to improve the paper with their valuable remarks and excavation of some early work references regarding binary rewriting.
Authors' addresses: Matthias Wenzl, FH Technikum Wien, Hoechstaedplatz 6, Vienna, 1200, Austria, wenzl@technikum-wien.at; Georg Merzdovnik, SBA Research, Favoritenstrasze 16, Vienna, 1040, Austria, gmerzdovnik@sba-research.org; Johanna Ullrich; Edgar Weippl, SBA Research, Favoritenstrasze 16, Vienna, 1040, Austria, CDL-SQI, TU Wien, Austria, JUllrich@sba-research.org, eweippl@sba-research.org.

Emulation. An emulator is a software or a hardware component that mimics the behavior of a platform on another platform[1]. During emulation, binary rewriting is used to translate requests and the respective responses from one processor architecture to another as it is done in virtualization software like QEMU [19]. Something similar is performed by Wang et al. [143] who use dynamic binary translation to offload time-consuming parts of a program to a more powerful computation node with a different processor architecture.

Observation. Observing programs during execution is the task of profiling and tracing tools that embed their monitoring code into the binaries under observation, where the latter resides in memory. They are used to find memory leaks (e.g., Valgrind [138]), monitor the adherence to specification (e.g., Pebil [82]), or for reverse engineering of data structures (e.g, Howard [123]).

Optimization. In high availability environments, such as telecommunication networks, down times have a major impact on service availability. Furthermore, the identification and optimization of timing anomalies in shared memory multiprocessor systems, like a high number of cache faults due to misalignment is of interest in high performance computing domains [86], where programs tend to run over a long period. In such situations, binary rewriting is used for run-time patching, as it is done by DynInst [25, 48], or Jennings and Poimboeuf [67].

Hardening. The absence of legacy build tools, build tool features, third party source code, or vendor support, introduces the necessity of binary rewriting at post-compile and link time [102]. This is of special interest if the utilized build tool originally used to create the binary lacks features such as modern exploit mitigation techniques. Specifically the insertion of stack canaries[2] [41] or address layout randomization [133]. Additional exploit aversion techniques applied at run-time, like attack recognition, also fall in this category (e.g., Zhang et al. [156]).

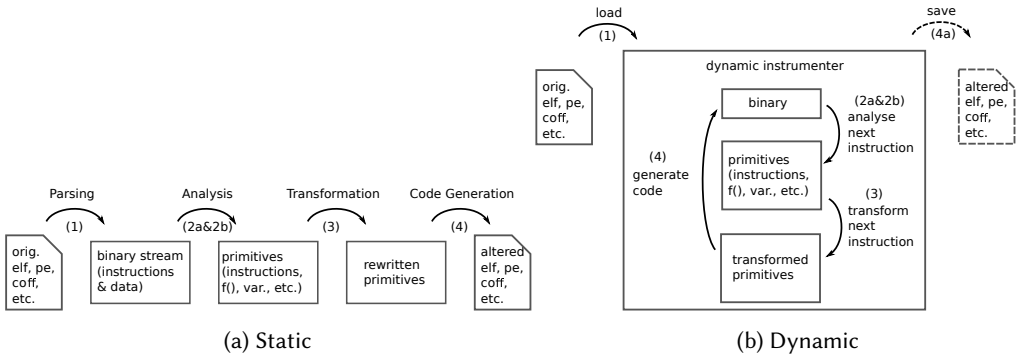
Throughout this paper, we will give an in-depth overview on the development and state of the art in the field of binary rewriting. After providing insights on the general approaches and their operation in principle, as well as a publication time line on binary rewriting tools in Section 2, we will start an in-depth investigation of the identified building blocks. Hereby, we will focus on the base techniques enabling various binary rewriting schemes that can be found in the 67 tool examined for this publication. Furthermore, will deal with the challenges and solutions of the particular problems in the binary rewriting process starting from the analysis step addressed in Section 3. Based on these insights, Section 4 contributes a detailed investigation of the current state of the art in binary code transformation techniques on behalf of the tools utilizing them. Afterward, Section 5 covers means to integrate the undertaken changes into the binary of investigation, permanently, if desired. Based on these results, Section 6 provides a detailed categorization of the investigated publications with respect to the findings in sections 3 to 5. Finally, Section 7 concludes the paper.

To the best of our knowledge, this work is the first paper giving a comprehensive overview on the techniques, challenges and solutions in the field of generic binary rewriting. In contrast, related work focuses on special aspects or fields of application regarding binary rewriting.

Cifuentes and Malhotra [32] provide a comprehensive comparison of static and dynamic binary alteration tools between 1987 and 1995. The authors advocate that intermediate representation will be a key technique to develop competitive dynamic binary translation tools. Additionally, Larsen et al. [79] conducted a survey on automated software diversity approaches, which when done at post-compile or run-time utilize binary rewriting to apply security related augmentations to

[1] https://www.kb.nl/en/organisation/research-expertise/research-on-digitisation-and-digital-preservation/emulation/what-is-emulation

[2] Stack canaries detect if someone tampered with a function's return address [41].



(a) Static (b) Dynamic

Fig. 1. Required steps to apply binary rewriting in principle.

software, such as adding stack canaries or implementing address layout randomization schemes. In contrast, our paper focuses on binary rewriting techniques themselves rather than single application domains. Within the context of this paper, software diversity would fall in the hardening domain. In fact, binary analysis shares an intersection with binary rewriting when it comes to preparing steps like disassembly and structural recovery. The work of Shoshitaishvili et al. [121] from 2016 gives an in depth overview on the current challenges and resolving strategies when doing binary analysis with automatic exploitation in mind. Andriesse et al. [8] did something similar, but with a focus on comparing recent disassembler frameworks. The authors concluded that while function start address detection is still not perfect, linear disassembly strategies are already able to deliver very good results under the assumption that the binary under investigation has been compiled with a recent compiler. Nanda and Chiueh [95] present a survey on virtualization techniques, including a coarse description of instruction rewriting, which is used in emulation to translate binaries between different processor architectures during run-time. In 2011, Hazelwood [59] conducted a survey on dynamic binary rewriting techniques. Nevertheless, to the best of our knowledge our work is the first to cover the whole process of binary rewriting from disassembly to code generation covering static and dynamic approaches with respect to 67 publications between 1992 and 2018. Furthermore, we provide a mapping between the base techniques enabling binary rewriting and the reviewed disseminations implementing binary rewriting tools.

2 BINARY REWRITING FROM HIGH ORBIT

The process of Binary rewriting modifies a given compiled and possibly (dynamically) linked program in a way that it remains executable without access to the source code for recompilation [81]. In general, binary rewriting can be classified into static and dynamic schemes. Static binary rewriting approaches operate on the binary while it is stored in persistent memory. On the contrary, dynamic binary rewriting is performed while the program of interest is executed.

2.1 The four steps to binary rewriting

Static and dynamic binary rewriting attempts can be split into four steps (see Fig. 1). First, the rewriter hast to retrieve the right information from the binary under investigation (see *Step 1: Parsing*). Subsequently, *Step 2: Analysis* performs binary analysis to recover the program's structure that has been lost during compilation and assembling. *Step 3: Transformation* uses the recovered information to alter the binary according to the user's requirements. *Step 4: Code Generation*

reintegrates the changes into the program. In the following, we provide an overview on the process of binary rewriting with respect to static (see Section 2.2) and dynamic (see Section 2.3) approaches.

Step 1: Parsing. Independently of its actual implementation, each executable format consists of *administrative* (e.g., section information) and *payload* data (e.g., instructions and global variables). The focus of binary rewriting is in obtaining and manipulating the *payload* data. Unfortunately, this information is usually scattered throughout the file in different ways. Furthermore, instructions (when using CISC architectures) and variables are not present in a delimited form and it remains unclear where one ends and a new one begins. Instead, instructions and global variables are present as a *raw binary stream*, grouped into different sections. Moreover, binaries have no type information stored with their (global) variables, thus the actual data type of a referenced address must be recovered separately. Hence, the purpose of this step is to obtain the raw instruction stream from the executable and pass it to a disassembler. Furthermore, the address ranges of the global variables and the content of the data section are retrieved for further analysis. As will be discussed in Section 2.3 and Section 2.2, mature solutions to solve step 1 exist. Therefore, we will focus on steps 2, 3 and 4 in Section 4 and Section 5.

Step 2: Analysis. The analysis step recovers the structure of a program's source code. While most of this structure is already lost during the compilation, assembling and linking process, stripping binaries from symbols removes even more information like names, addresses and types. At the end of step 2, the binary of interest is disassembled, the recovered instructions are grouped in functions, identified variables are associated with an appropriate data type and a control flow graph (CFG) has been generated.

Stripped binaries, which are the result of step 1, are agnostic towards high-level concepts like functions, data types and even clearly separated instructions. Thus, all information that remains before analysis is unstructured binary code placed at specific sections in an executable. Nevertheless, since we are in step 2, the raw instruction stream as well as the global variable from the data sections are already obtained from the binary of interest. Therefore, the analysis process is two-fold:

First, the raw instruction stream is dissected and decoded by the disassembler. At the end of this step, each bit pattern identified as instruction is available as its assembler representation together with its parameter, referenced memory addresses and location in the binary. By considering all kinds of branch instructions within the raw binary stream, a control flow graph of the binary can be generated. A control flow graph is a directed graph with the nodes made up of (discovered) basic blocks. The graph's edges identify the branch source to branch target relations [4]. Each basic block consists of a series of instructions ending with a branch instruction. Considering the analysis step, CFGs are a common data structure to store the analysis results.

Second, as much as a binary does not have an idea of data types and instruction boundaries, it is also agnostic towards the concept of functions. Thus, it is the task of function recovery algorithms to find and group series of instructions connected by branches to function blocks, as well as to determine the function's entry and exit points.

Step 3: Transformation. Once function boundaries and a control flow graph have been recovered, the binary of interest is ready to be altered. Alterations can be done at *instrumentation points*. Instrumentation points are user specified locations in a binary where (a) the control flow changes, e.g., the rewriter wants to redirect the control flow to a new set of instructions; (b) instruction changes can be applied, e.g., the rewriter wants to augment every function with certain profiling code.

The amount and location of the instrumentation points are directly related with the general binary rewriting approach, which can be either coarse or fine-grained. Coarse-grained approaches

generally allow for persistent binary changes (executable on disk is changed), but only at branch locations. In contrast, fine-grained attempts are able to alter every instruction in a binary at the price of high overhead .

Step 4: Code Generation. The last step integrates the intended changes into the binary of interest in such a way that it stays executable. In general, there are three possibilities to do so: (1) A new section holding the changes is carved in the binary file of interest. This includes the need for detours at instrumentation points making the new code reachable during execution. (2) A detour is added during program execution making the new code reachable while the program executes, but leaves the binary on disk in its original state. (3) An arbitrarily altered binary is fed to a commercial off-the-shelf assembler that creates a completely new executable.

2.2 Static binary rewriting in principle

Static binary rewriting as shown in Fig. 1a operates on files stored in persistent memory. Therefore, in step *(1)*, retrieving a raw binary stream consisting only of instructions and global variables from a given executable must be accomplished. However, the program's instructions and global variables of interest are likely scattered around the file in several sections interleaved with administrative information. For example, the Executable and Linking File format ELF [152], commonly used in modern Unix based operating systems, holds at least an ELF header table and a section header table as auxiliary information, which are not of interest during disassembly (step *(2)*). The same kind of administrative information is available in the Common Object File Format (COFF) [135], used for example by Texas Instruments, or the Portable Executable (PE) [105], utilized by Microsoft. Nevertheless, to accomplish step *(1)* a plethora of libraries such as BFD [129] as a part of the binutils package are readily available to parse and convert an executable into a binary stream that can be processed by a disassembler. The obtained binary stream is then fed into a disassembler (Fig. 1a (2a)), returning a set of detected instructions and data together with address information. Afterwards, structural recovery algorithms are used to build control flow graphs, extract function start addresses and recover data types (Fig. 1a (2b)). The detected structures of interest are subsequently used as input for one of the following static binary rewriting approaches (Fig. 1a (3)).

Direct. The oldest static binary rewriting scheme operates directly on the instructions of interest. Although this approach works well when altering instructions of similar length and semantic consequence[3], in the worst case adding or removing instructions implies an update of all branch targets [130].

Minimal-invasive. When utilizing this approach a new section is inserted into the binary holding the intended instrumentation code. In order to reach this section, unconditional branches are inserted at instrumentation points within the original program flow.

Full-translation. These schemes require the complete binary to be transformed into an intermediate representation allowing alterations on instruction granularity. Intermediate representations in the domains of compilers, binary analysis, and binary rewriting are data structures that allow for hardware architecture independent description of programs without loss of information [31]. The process of transforming a more detailed representation into a more abstract representation is known as lifting [15] (E.g., lifting x86 instructions to LLVM's intermediate representation.) .

Finally, the altered binaries must be reassembled and written to disk for the changes to take effect (Fig. 1a (4)). When using minimal-invasive rewriting, the reassembled binaries differ only at the instrumentation points and at the newly added section holding the instrumentation code

[3]When altering MIPS instructions, some introduce a required delay slot.

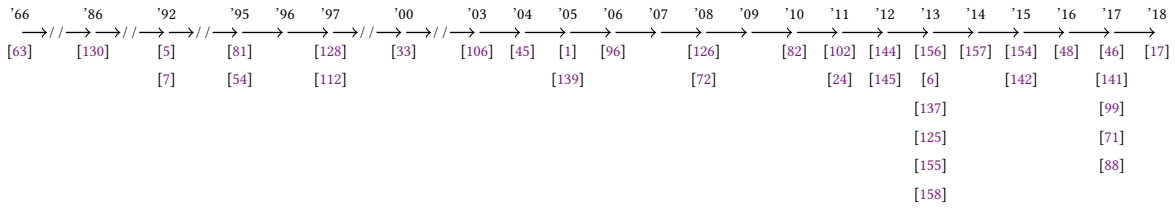


Fig. 2. Timeline of publications presenting static binary rewriting tools spanning from 1966 to 2018. (The publications [1, 46, 49, 156] present mixed approaches, therefore they are placed in both timelines.)

from the original binary. In contrast, a reassembled full-translation binary may have a completely different layout, although only the same changes as in the minimal-invasive attempt might have been made. This difference is caused by the semantic equivalence approach used in common lifters for a detailed explanation of the problem and its consequences.

A timeline of publications regarding static binary rewriting approaches is depicted in Fig. 2. Within the years 1966 to 1997, only few papers and tools concerning static binary rewriting were published. The interest for this approach increased in 2001, was further raised in 2010 and peaked in 2013.

2.3 Dynamic binary rewriting in principle

Dynamic binary rewriting performs the analysis and transformation operations during program execution. In contrast to static binary rewriting, that executes all 4 steps in a row, dynamic binary rewriting implements an iterative algorithm as shown in Fig. 1b.

Thus, upon the first step the application of interest is loaded into an instrumentation program that is similar to a debugger, as it is able to monitor each executed instruction and accessed variable (Fig. 1b (1)). This can be done using the PTRACE API under Linux [103],the application debugging API under Windows [69], or a custom loader as it is done by Pin [84], or DynamoRIO [23]. During execution, the binary is disassembled along the paths that are covered by its input data (see Fig. 1b (2a, 2b)), hence obtaining the structures of interest at run-time. Just as in the static rewriting approach, binary transformation is applied to the structures of interest (see Fig. 1b (3)), which in turn have to be integrated into the program flow. Since it is hardly possible to disassemble the whole binary within a single run (due to path coverage limitations), either only the alteration of covered paths is of interest (e.g., [23, 48]), or techniques to improve the path coverage are applied [53]. Depending on the rewriter, program alteration is either temporary (see Fig. 1b (4)) or persistent (see Fig. 1b (4,4a)).

While non-persistent alterations as performed by Valgrind [98] only suffer from the run-time transformation overhead and disappear once the program terminates, persistent solutions such as presented by Hawkins et al. [58] also keep parts of the original binary within their rewritten image to maintain their functionality. Therefore, persistent dynamic binary transformation induces time and memory overhead during run-time.

A timeline of publications related to dynamic binary rewriting is shown in Fig. 3. After an initial publication in 1987, the interest in dynamic binary rewriting increased in the mid 90's, 2000 and 2011, with a peak in 2017, after a short intermission between 2008 and 2010.

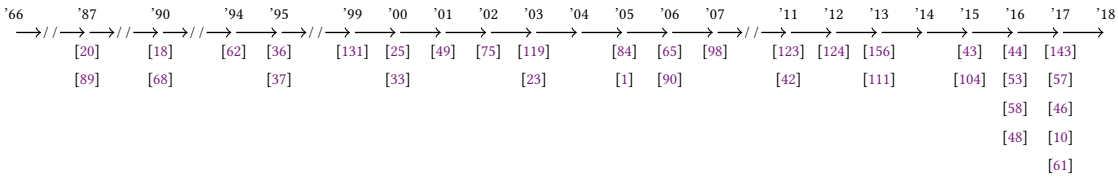


Fig. 3. Timeline of publications presenting dynamic binary rewriting tools spanning from 1966 to 2018. (The publications [1, 46, 49, 156] present mixed approaches, therefore they are placed in both timelines.)

3 ANALYSIS

The purpose of the analysis step is to provide information on the building blocks of a binary to enable the subsequent transformation step, which performs the alterations on the identified instrumentation points. This includes the tasks *disassembly*, *structural recovery*, as well as *label, symbol and data type* extraction. Although most of the herein described techniques work well on non-obfuscated binaries, the respective authors mention that the contrary is true for binaries containing even simple obfuscation techniques. Hence, we omit to consider obfuscated binaries.

Disassembly: The task of a disassembler is to *(1)* dissect the incoming raw binary stream (see Section 2) into single instructions as well as to *(2)* decode these instructions and map them to a mnemonic and its associated parameters. For example, the byte-wise interpreted bit pattern *0x89 0x86* is identified and decoded as "copy the contents of register *ecx* to register *eax*" mnemonic ("`mov eax, ecx`") on an x86 architecture. The actual decoding step can be supported by libraries such as Intel XED [66] that provide instruction code to mnemonic mappings targeting x86. While decoding a single instruction is a straightforward task, identifying the borders between two succeeding instructions is more challenging when utilizing CISC instruction set architectures due to their variable length instructions. Consequently, elaborate disassembly strategies have been developed to overcome these limitations, as shown in Section 3.1.

Structural recovery: Detecting functions and obtaining control flow graphs are the main tasks of structural recovery algorithms. However, from an assembler level perspective, none of this structural information exists in a binary. **(a)** Control flow graph information is implicitly encoded into branch instructions and must be recovered by following branches from their origin to their target address. Following branch targets is a challenging task, especially considering indirect branches. **(b)** Function borders (entry and exit points) may depend on the used compiler version and optimization level making them hard[4] to detect with simple pattern matching approaches. Therefore, this section discusses attempts to improve function detection rates.

Label, symbol and data type extraction: Knowing that a global variable or a constant is a pointer type instead of an integer or floating-point number is important (e.g., when recovering indirect branches). Under certain assumptions, distinguishing between pointers, integers or floating-point numbers when analyzing a stripped binary can be performed by UROBOROS [142]. An investigation of the general problem on performing type inference on binaries is provided by Caballero and Lin [26].

[4]Simple pattern matching approaches require the exact function entry/exit conventions to be in place. Thus, even simple deviations, which happen at certain optimization levels (e.g. function in-lining and prologue/epilogue duplicate removal) can cause these detection mechanisms to miss a function [106].

3.1 Disassembly

The primary tasks of a disassembler are *(1)* to distinguish between code and data and *(2)* to decode an identified instruction at a specific address into its corresponding mnemonic together with its parameters. Therefore, the result of the disassembly step is a separated set of recovered and identified instructions.

General disassembly challenges: Resolving whether a part of the binary stream is an admissible instruction or part of in-line data is undecidable in the general case [64, 146]. This is mainly caused by *(a)* today's dense instruction sets (e.g., almost any bit combination may reveal a valid instruction) and *(b)* variable length instruction set architectures. In-line data can be caused by the compiler placing constants (e.g., when building jump tables) in the text section to speed up program execution in unified cache architectures [52]. A given bit pattern of in-line data in the binary stream can be easily mistaken for a valid instruction, or vice versa. In variable length instruction set architectures (e.g., x86) instruction alignment is not required, thus the beginning, or end of an instruction is not always clearly visible. Nevertheless, the undecidability problem affects only *static* disassembly algorithms where *dynamic*[5] disassemblers suffer from the path coverage problem [62]. Here, the disassembly path of a program depends on the given input data and would require exhaustive testing[6] to reach a 100 percent code coverage [94]. Additionally, binaries are available as stripped (without symbol information) and unstripped (including symbol information) versions, introducing or removing additional hurdles for the disassembler [110].

Furthermore, in order to overcome some of the disadvantages of static approaches, dynamic disassembling strategies are discussed in Section 3.1.3. Additionally, hybrid attempts utilizing the strengths of the aforementioned disassembly techniques to cover up their weaknesses are described in Section 3.1.4.

3.1.1 Static - linear sweep. Linear-sweep traverses a binary sequentially from a given entry point trying to interpret the binary stream as instructions. While this approach is known to yield good results regarding the ability to find and decode instructions occurring in a binary [8], it lacks the ability to detect in-line data reliably (e.g., pointer constants embedded in the text section).

An approach to mitigate the in-line data problem in CISC (e.g., x86, x86-64) architectures has been developed by Bauman et al. [17] by applying an iterative linear sweep disassembler. Here, the disassembling process begins from a section's start address and ceases when the disassembler is not able to decode any further instructions. Now, the algorithm restarts with an offset of one from the start address. This offset increasing scheme continues until the whole section is disassembled. Multiple recovery of instructions is avoided by skipping already decoded instructions. This allows for completely disassembling a text section including in-line data on CISC architectures.

3.1.2 Static - recursive traversal. Recursive traversal algorithms implement linear sweep until they hit a branch. Then the branch target is taken in either depth-first or breadth-first search manner [106] and disassembly continues. The motivation behind recursive traversal is to avoid in-line data by jumping only to valid instructions.

However, branch targets might be calculated at run-time, therefore some paths may be missed by recursive traversal. Such constructs are called indirect branches.

A special case of indirect branches operating on arrays of function pointers is implemented by *jump tables* [115]. Their jump targets are calculated at run-time, which is done using a variable offset to a constant base pointer on a function pointer array. In fact, the boundaries of the function pointer

[5] Disassembly at run-time

[6] *Exhaustive testing* stems from the software testing domain, meaning that all paths in a computer program must be covered by the test cases, which is clearly intractable in any reasonable sized program [94].

array must be determined correctly by the disassembler. Incorrect array length determination may lead to *(a)* an incorrect control flow graph or *(b)* mistaking data for instructions leading to incorrect basic block identification in the structural recovery step.

3.1.3 Dynamic. In order to avoid the drawbacks of a static disassembler, dynamic disassemblers may be employed. The issues considering indirect branches revealed in Sections 3.1.1 and 3.1.2 are omitted by disassembling the program during execution. However, as the covered paths of a disassembly run highly depend on the chosen input, exhaustive testing would be necessary to reach full path coverage. This is clearly illusive in reasonably sized programs [94], therefore being the biggest drawback of dynamic disassembly approaches. Nevertheless, when dynamic disassembly algorithms are employed, the complete disassembly of a program is seldom of interest.

However, mitigation techniques as rollback and forced execution can be employed to further increase code coverage [151]. Here, the complete program is traversed in depth-first search order depending on the given input of the program. At each conditional branch decision, the program state is saved before the execution proceeds. Once the first leaf on the program's branch tree is detected, the state of the last visited branch point is restored and its alternative path is taken by the disassembler. Thus, increasing path coverage. In order to increase search performance, already covered paths are marked and are not disassembled again.

3.1.4 Hybrid. Additionally, hybrid approaches to overcome the mentioned deficiencies of dynamic and static approaches as implemented in Nanda et al. [96], using a mixture of dynamic, linear-sweep and recursive traversal to obtain a disassembled binary. Here, disassembling starts with a static algorithm and resorts do dynamic disassembly when needed. Furthermore, static speculative approaches as presented in Kruegel et al. [78] aid in finding run-time computed or indirect branch targets, which are not clearly visible to the disassembler. Such constructs can be seen in jump tables, position independent code (PIC) [150] and virtual function tables [144].

For an in-depth evaluation of selected open source and commercial disassemblers see Andriesse et al. [8]. Furthermore, note that *objdump*[7] or *capstone*[8] implement a plain static linear approach in which most other disassemblers realize mixed static approaches including *radare2*[9], *relyze*[10] and *IDA-pro*[11]. Dynamic disassemblers are implemented by *DynInst* [48] and in the work of Kiriansky et al. [75].

3.2 Structural recovery

Disassembled binaries are agnostic towards concepts like *control-flow graphs* and *functions* (and data types). However, these structural information (or disassembly primitives) are vital to perform thorough binary rewriting [8, 21, 91, 113, 137, 144]. Therefore, this section discusses approaches to retrieve the structural information from a disassembled binary without symbol information.

Program (binary) slicing: In the context on binary rewriting, binary slicing, a particular form of program slicing is used in *indirect branch resolution*, *function boundary detection* and *variable/data type recovery*, making it a central analysis technique in the step of structural recovery. In general, static [148] or dynamic [77] program slicing conceptually works by concentrating only on the alteration of specific variables within a program's flow in order to determine its local meaning in a greater context by masking out all information not directly associated with the variables

[7] https://sourceware.org/binutils/docs/binutils/objdump.html
[8] http://www.capstone-engine.org/
[9] http://rada.re/r/
[10] https://www.relyze.com/overview.html
[11] https://www.hex-rays.com/products/ida/index.shtml

of interest [22]. Program slicing is a fundamental technique when analyzing and understanding complex software [149]. Binary slicing is a special application of generic program slicing that operates on instruction level rather than high-level language statement level, as it is done in classic program slicing [34].

Control-Flow Graph: A common data structure to have all recovered disassembly primitives in one place is the control flow graph (CFG), a directed graph with the nodes made up of discovered basic blocks [4]. A perfectly recovered CFG in the terms of the analysis step is a weakly connected graph[12]. Hence, all indirect branches must be resolved. In short, indirect branches are generally resolved during program execution making it hard[13] to succeed for static analysis methods and cumbersome[14] for dynamic approaches. In contrast, direct branches are resolved at compile time, thus their jump targets are known in advance. Therefore, resolving the indirect branch resolution problem means solving the control flow graph recovery problem [121, 151]. Furthermore, an ideal CFG is augmented with a complete symbol table providing function start addresses and variable addresses (and data types). However, this would require an exhaustive symbol table to be at hand, which is rarely the case [113]. Techniques such as *pattern matching* and *heuristics* [106], *binary slicing* [34], abstract interpretation (e.g., *value set analysis* [13], *simple expression tracking* [52]) are employed to fill the gap when using static structural recovery methods. Dynamic approaches benefit from ideas like *code-caches* [23] and *forced-execution* [151] to perform their tasks.

Function recovery: Almost all basic blocks in a binary are assigned to a corresponding function[15]. Every function is supplied with its start and end address as well as function parameter information. In general, application binary interfaces specify calling conventions that are followed by compilers, thus reducing function start address detection to a pattern matching problem. However, different compiler versions implement different conventions under varying optimization levels making pattern matching a cumbersome task.

3.2.1 Binary slicing. Just like program slices, binary slices can be computed in a *forward* and *backward* manner. Forward (binary) slicing is used to answer questions like "*What statements (instructions in binary context) are affected by the value of variable v at statement (instruction) s?*". In contrast, backward slicing tries to solve queries of the type "*What variables v have been affected by executing the program up to point (address) p?*". In the context of structural recovery, the latter can be used to approximate the admissible value interval a register might have in order to determine the target address range for an indirect branch. Binary slicing algorithms can be applied to stripped binaries.

Furthermore, binary slicing is available as *(a) intra-procedural* binary slicing and *(b) inter-procedural* binary slicing. *Intra-procedural* binary slicing has been developed by Cifuentes et al. [34] and is limited to computing forward and backward binary slices within a given procedure (function). It implements a static slicing algorithm based on the *goto* slicing algorithm developed by Agrawal [2]. *Inter-procedural* binary slicing has been developed by Kiss et al. [76] and is an extension to the work by Cifuentes. It is able to compute backward and forward binary slices across function boundaries using the intra-procedural algorithm in its core.

[12] A weakly connected directed graph has an underlying connected undirected graph

[13] In the worst-case scenario, an indirect branch may hit any address within the text section/segment of a program. Furthermore, the indirect branch target address may be computed from user input making the decision on the actual indirect branch target address undecidable in the worst case [8].

[14] Since indirect branch targets may be taken more the once in a program flow, the overall dynamic analysis overhead can be reduced when already disassembled basic blocks are not disassembled again. Hence, various caching and indirect branch target address prediction schemes exist that re discussed in Section 4.2.

[15] An exception would be *reset code* that is immediately executed after power on.

3.2.2 Indirect branches. An indirect branch is fully defined through a generic register expression of the form base + index × scale + offset, with *base* and *index* being registers [13]. Nevertheless, only the *base* address register is mandatory (e.g., "`jmp eax`" is a sufficient indirect branch expression in x86 assembly).

In contrast, a direct branch can be identified by a branch target address that is directly encoded in the instruction, for example, the x86 instruction "`jmp 0xdeadbeef`" realizes an unconditional direct branch. Considering CISC architectures, a direct jumps target address can always be reached within a single instruction [125]. However, if the direct branch is implemented on a RISC machine, this might not always be the case since branch instructions on RISC machines are not able to cover the architecture's whole address space.

Furthermore, branch target addresses of indirect jumps are not directly visible for static disassemblers. Additionally, the value of an address register may stem from a computation involving an arbitrary user supplied value that cannot be resolved directly [46].

Indirect branches can be resolved using static backward binary slicing. However, binary slicing operates on registers only and is not capable of resolving indirect branches of the type "`jmp [eax]`". Here, the branch target is stored at the memory address to which *eax* is pointing (e.g., a global variable). A tool implementing binary slicing would have to fall back on either marking the indirect branch as not resolvable, or employing a dynamic recovery approach. EEL [81] as an early binary rewriter covering this subject implemented the dynamic indirect branch resolution approach for jumps of type "`jmp [eax]`".

Tracking target addresses of indirect branches referencing global and local variables can be achieved using abstract interpretation based approaches [74]. Abstract interpretation provides means to link semantics with different levels of details of a program in order to reason about its run-time behavior [40].

A look up table based approach using static pre-computation and dynamic resolution, based on Pin's [84] indirect branch resolution scheme, has been proposed by Bauman et al. [17]. Here, the authors pre-compute a lookup table holding the address of every byte in a given section. Each entry of the look up table is filled with the new address of the primitive relocated during the transformation step, thus creating an associative array. For this to work, every indirect branch must be transformed to a direct branch[16] at the right location of the look up table. If the entry is valid, the new branch destination address can be obtained, otherwise a segmentation fault occurs.

Static - Value Set Analysis: Value set analysis (VSA) implements an abstract interpretation approach to find a set of over-approximated values for each *data object* at a given program point [13] (e.g., what is the possible value of register *ebx* at location *0xdeadcode*). A key feature of VSA is that it tracks integer-valued and address-valued quantities simultaneously [14]. The technique has similarities with pointer analysis, but aims at assembly level programs.

In VSA, a *data object* might be a register, a global variable, a local variable or heap allocated memory. Each *data object* (i.e., global variable) is represented by an address location, called *a-loc*. An *a-loc* is defined by a tuple consisting of a *range* (*rng*) and *offsets o* of the form $\{rng \mapsto o\}$. An instance of a range might be a specific global variable. In order to provide a bounded over-approximation of the possible values of an *a-loc*, *offsets* are realized as strided intervals with lower and upper bounds as well as a size information (stride[lower, upper], size).

Each *a-loc* can be referenced from within various procedures. Now, the idea is to collapse all different occurrences of a *range* with its different *offsets* into the form $\{\text{rng} \mapsto \{o_1, o_2, ..., o_n\}\}$, making the over-approximation and bound estimation possible in a convenient way. The convenience lies in the fact that due to the combined procedure local *a-loc* representations a global bound computation

[16] Plus some administrative code; For details see section IV of Bauman's paper [17].

can be done efficiently. Furthermore, the different *ranges* are mapped into a *value-set* of the form: $\{rng_1 \mapsto \{o_1, o_2, ..., o_n\}, ..., rng_r \mapsto \{o_1, o_2, ..., o_n\}\}$.

Let us assume[17] the instruction "`jmp 0x1000[eax*4]`" has the current *a-loc* representation for the register *eax* of $\{eax \mapsto \{[0, 9], \bot, ..., \bot\}\}$, with $\bot$ denoting the empty set. In this scenario the algorithm is interested in computing the possible target addresses of the indirect branch given the above-defined *a-loc* for *eax*. An evaluation of the *jmp* instruction under *eax*'s *a-loc* reveals the possible branch targets of $\{0x1000, 0x1004, ..., 0x1036\}$ that can be added to the CFG. However, in case the abstract interpretation of the branch instruction would result in a set of all possible target addresses, no edges would be added resulting in a possible under-approximation of the indirect branch's target addresses.

Currently, VSA based approaches are, for example, used in rewriting tools such as Bitblaze [126] and BodyArmor [124].

A further application of VSA is implemented in Jakstab [72, 73]. Here, abstract interpretation is used to compute an over-approximation of safe values for indirect branches, by utilizing data flow analysis on an intermediate language. The bounded address checking part of the approach used to check the sanity of the approximated values is inspired by VSA.

Another approach using symbolic execution [15] as a basis has been used in the tool JITR [38], that is able to recover indirect branches occurring in switch/case statements. The authors introduce a value set analysis approach embedded in a symbolic execution context. Another SVA approach, which mixes dynamic and static value set analysis (VSA) is employed by Mayhem [28] for automatic exploitation purposes.

Static - Simple Expression Tracking: A different approach leveraging the transformation capabilities of LLVM [83] is used by Di Federico et al. [46]. Here, the binary of interest is lifted to LLVM's intermediate representation using QEMU's tiny code generator lifter as a preliminary step. The obtained LLVM program is then transformed into single static assignment (SSA) representation. Based on SSA, the author's simple expression tracker (SET) in conjunction with shift offset range data flow analysis is used to determine an over-approximation of possible jump targets.

A SET computation starts by detecting an indirect branch. Now, all non-constant operands (i.e., registers, memory locations) are tracked back until a constant is found. Then, the constant is used as input for the computation of the true target address of the indirect branch.

In order to supply bounded values for SET's results that depend on non-deterministic data, such as user input, SET is accompanied by a data flow analysis based approach called offset shifted ranges analysis (OSR). OSRA is able to provide bounded values for expressions of the type constant + a × b for small values of *a* and *b* [46].

Dynamic indirect branch recovery: In opposition to static approaches, dynamic structural recovery algorithms are easily able to resolve many branches correctly, but suffer from the same problems as static approaches regarding function boundary detection and data type retrieval. Furthermore, the code coverage issue is still existent, which can be tackled with *forced execution* and *rollback* [151]. Here, Xu et al. implement a first-depth search based approach that saves the location and state of the program before executing each branch. Upon reaching a leaf in the dynamically extending CFG, a rollback to the last branch including an execution of its alternative path is forced, leading to improved CFG recovery.

Unfortunately, dynamic branch recovery approaches suffer from high computation overhead [23] that can be reduced by utilizing *code caches*. Indirect branches are processed by translating their original addresses to their corresponding code cache address every time they are hit using a hash

[17] Taken from Balakrishnan and Reps [13].

table. The code cache mechanism provides the ability to detect already disassembled basic blocks and directly link them together as they are processed by branching resulting in a processing speed up [23].

3.2.3 Function recovery. Function recovery consists of *function boundary* and *function start* detection. Where function start address identification tries to determine which addresses resemble a function's entry point, function bound detection retrieves the first and the last address of a function [9]. To know the bound of a function is vital for a variety of further analysis and rewriting techniques like control flow protection [158] and binary instrumentation [82].

In the common case, function boundaries are determined by function prologues and epilogues. Both usually follow a pattern of certain instructions defined by the calling convention in the application binary interface (ABI). The next paragraphs provide an introduction on commonly used function recovery strategies.

Pattern matching and heuristics: A direct approach for function start address detection is realized by using pattern matching and heuristics [113, 146]. However, as shown by Andriesse et al. [8], the function start detection success rate drops to about *80%* with increasing false positives caused by handwritten assembly, or encountering an optimizer's results that are not covered by the pattern matching/heuristic schemes. Furthermore, pattern matching approaches are highly compiler dependent [8].

This issue can be mitigated to some extent using machine learning based approaches as done in Byteweight [16] or by Shin et al. [120]. While Byteweight performs function bound detection by utilizing an oracle providing the required mappings between function prologues (epilogues) and symbol information in combination with value set analysis to obtain function bodies, the approach by Shin uses trained neural networks to perform function bound detection. Although, better than simple pattern matching, machine learning based approaches are highly dependent on the quality of the training sets (i.e., which compiler types, version and variety of binaries).

Pattern matching in combination with backward binary slicing, used for indirect function calls, is performed by Qiao and Sekar [108]. After obtaining the slice for an indirect function call, pattern matching is used to determine whether the branch is a function call or an intra-procedural branch.

Graph based approaches: A true compiler agnostic, but still architecture dependent approach for function start address detection for x86 has been developed by Andriesse et al. [9]. The approach works by creating a global call graph[18] of the binary, followed by a *weakly connected component* analysis: First, all edges created by a direct *call* instruction are hidden. This creates a partitioned call graph, where each partition consists only of basic blocks connected through instructions other than *call*. Now, all targets of the direct *call* instructions are used as function entry points to determine the neighborhood of the function candidate by following the control flow regardless of direction. While doing so, return instructions are not of interest. These steps will detect all directly called functions ceasing the *weakly connected component* analysis. The remaining functions are located in the spare basic blocks not yet associated with a function and are detected by further applying *weakly connected component* analysis and function entry point detection using global control flow analysis.

In addition to the compiler agnostic approach by Andriesse et al. [9], Federico and Agosto. [52], as well as Federico et al. [46] introduced an architecture agnostic method for function boundary detection. The approach uses QEMU's tiny code generator as a first step before lifting the program into the LLVM intermediate representation where the detection mechanism is executed. The

[18] A call graph is a CFG without basic blocks. A global call graph consists of all edges within the binary regardless of possible function affiliation [158].

algorithm starts from candidate function entry points. These are obtained by harvesting the programs LLVM representation including global variables with respect to *(a)* direct function calls and *(b)* global constants. Furthermore, only global constants that refer to already recovered basic blocks if they are interpreted as pointers are taken into account. Now, each targeted basic block (from either *(a)* or *(b)*) is explored by following other branches until a function return pattern is reached. The algorithm stops and returns an over-approximation of a candidate's function bounds. In the next step, the set of candidate start addresses is filtered by keeping all start addresses that are exclusively reached through direct calls or through skipping jumps (e.g., through tail calling).

3.3 Label, symbol and data type extraction

Disassembled and with respect to data and instruction classified stripped binaries have no concept of labels, symbols and data types [3]. However, their presence is important in certain analysis and code generation scenarios [142]. Therefore, symbolization and data type recovery algorithms are used to augment disassembled binaries with the required information:

Labels: In general, labels are used as a placeholder for branch targets that are resolved to addresses during assembly when it is clear how much memory the instructions between a branch source and its target will consume. Since the transformation step in which the binary is eventually modified (see Section 4) is likely to invalidate the original branch target addresses, re-labeling is necessary when feeding a transformed binary to a commercial off-the-shelf (COTS) assembler.

Symbols and data types: A disassembled program has no concept of *data types* or *symbols*. From an assembly level perspective four consecutive bytes in memory may represent a 4-character long string, a 4-byte integer, or a pointer on a 32-bit processor. However, when recovering indirect branches, it is important to know whether a global variable shall be interpreted as an integer, a base pointer of a jump table, or a string. A sub problem in data type detection is function parameter detection, where stack based recovery is easier than register based retrieval because stack based function parameter signatures are well defined [97]. Making these decisions and creating a symbol table of its findings is the task of a symbolization algorithm.

Symbolization: Pointer constants cannot easily be distinguished from numeric constants, or string constants in disassembled stripped binaries due to the lack of labels and symbol names. Hence, symbolization algorithms provide means to do the differentiation as well as the creation of a symbol table that can be used in later steps. Furthermore, these algorithms have the ability to detect certain data types such as strings and floating-point numbers [53, 57, 141, 142].

A straightforward attempt in doing so is to test whether machine word wide constants point within the valid address range of the binary when interpreted as a reference. This approach is feasibly possible since address boundaries of *.text* and *.data* sections are always contained in the executable when operating on unobfuscated binaries.

For example, let us consider a binary with the *.text* section start address of *0x8000000* and a length of *0x50000*[19]. In case the symbolization algorithm encounters the numeric constant of *0x00000300* at address *0x8060004*, represented as *0x00030000* on a 32-bit little endian architecture, it is obviously a constant since it is pointing outside of the *.text* section. However, in case a numeric constant of *0x804ec3d* is detected at address *0x8060080* it can either be a pointer resembling a branch target in the *.text* section or the little endian representation of the floating-point number *4e-34* declared as static variable.

[19]Example is taken from Wang et al. [141].

This leads to the necessity for more sophisticated symbolization approaches than pure range plausibility checking, as done by Uroboros [142], that is capable of differentiating between pointer, integer constants and strings (null and non-null terminated).

For example, Zipr's [53, 57] approach is based on address pinning, which is performed during (indirect) branch resolution in Zipr's dynamic structural recovery module. An instruction's location is marked for updating (pinning) during transformation as soon as it is referenced by an (indirect) branch. The algorithm keeps a record of what branch targets what address, making it possible to build a dependency graph that is used as update reference during rewriting. As soon as an instruction is moved due to instruction insertion, the update mechanism is triggered causing all references to be adapted in such a way that the dependency graph stays correct. Additionally, Zipr creates a symbol table holding all detected pointers.

An evolution of Uroboros' approach to data type identification is utilized by Ramblr [141]. Wang et al. employ a mixed approach using blanket execution [50] and localized value set analysis. Blanket execution is an abstract interpretation based semantic similarity detection algorithm using feature weighting. A feature is defined as a read or write operation to a memory location of interest and is used to detect dependencies between variables and constant definitions in order to determine their types. This is achieved by applying a combination of data dependence analysis, program slicing, and value-set analysis to address corner cases such as unaligned variables.

For example, in order to identify and recover a jump table, an intra-procedural backwards slice is generated with respect to the jump target. On behalf of the slice, value-set analysis is applied to find all entries within the table and to recover an over-approximation of possible branch target addresses. The algorithm terminates by marking the recovered table as pointer array.

Nonetheless, compiler optimizations such as base pointer re-attribution may lead to missed pointers. For example, the C expression "`arr[var - 'B']--;`" computes the actual index of the array *arr* by subtracting the constant offset *0x42* (the hex value of the ASCII character *B*) from the variable *var* to obtain the value stored in the array which is then decremented. Since the subtraction of *0x42* is always done, an optimizing compiler may replace the base pointer of *arr* with the result of the computation "`arr - 'B'`" which can result in the new base pointer being located outside of the admissible memory region stated by the binary meta data (e.g., *arr* is the first variable in the *.data* section). Therefore, a symbolization algorithm will mistake the pointer for an integer constant. This issue can be overcome by enlarging the address boundaries provided by the binary meta data with a small experience based margin. The enlarged memory region can now be used as a pre-filter to identify immediate values that can be symbolized according to Wang et al. [141].

4 TRANSFORMATION

The transformation step's task is to modify object files or already linked binaries in such a way that they include the desired additional functionality, but stay executable [81]. However, compilers produce densely packed code without empty address ranges except for padding within a section. This implies that there is no space for additional instructions after code generation. Hence, it is the task of the transformation step to find and exploit these locations within a binary where adding instrumentation code is admissible. Such locations are called instrumentation points [21]. Binary rewriting attempts can be generally categorized into *static* and *dynamic* approaches, that can be augmented with *minimal-invasive* and *full-translation* techniques.

Static: Static binary transformation performs alterations directly at the instrumentation point without any intermediate steps. It is most commonly used for link-time code manipulation and alterations affecting the instrumentation point only. One of the first static rewriters publicly available was the Liberator toolset for Honeywell's Series 200 systems presented

in 1966 [63]. The software's purpose was to translate programs from the IBM 1400 series machines to Honewell's own computer system.

Dynamic: Dynamic transformation is capable of performing alterations at instruction granularity during program execution. In order to support programs written for the MPE V operating system that were executed on HP-3000 processors for the new HP precision architecture, Hewlett Packard implemented one of the first a dynamic binary translation approaches in 1987 [20]. The tool was implemented as an emulator that was capable of resolving branch targets that could not be determined within a single transformation run (e.g. indirect branches).

Minimal-invasive: Minimal-invasive based transformations operate on branch granularity. They work by redirecting the program flow to a newly created part of the program in the binary under investigation. The additional instructions are inserted before the program flow is continued. ATOM [128] was a follow up tool on OM [5] by the same authors that relied on OM's transformation scheme, but new code is added in a distinct section between the text and data section, creating a first version of a static minimal-invasive binary transformation style. Since the transformation is applied before linking, a valid binary can always be produced. The actual editing can be done using the binutils bfd [129] library. The first static minimal-invasive transformation approach to allow for persistent binary alteration on stripped binaries was presented by Prasad et al. [106] in 2003.

Full-translation: Full-translation based transformations are able to transform binaries at any instruction, but require the binary of interest to be transferred (*lifted*) into an intermediate representation. The first tool implementing this approach is ATOM [128]. Since ATOM also implements the first version of a static minimal-invasive binary transformation scheme, it can be seen as a mixed approach.

4.1 Static

Static binary rewriting can be divided into approaches requiring symbol table information and those that do not [21]. However, many of the latter approaches are able to leverage symbol table information if available.

Rewriting without obligatory symbol table information. Altering instructions in a program's text section can have several implications.

In the best of all cases, static rewriting without symbol information is interested in exchanging instructions without the necessity to expand the text section of the program under investigation, thus avoiding the need to update branch target addresses. Such a case might be immanent when doing binary translation in fixed length instruction set architectures with semantic equivalent instructions that differ in their encoding as it has been in done by Honeywell [63] in 1966. Honeywell's liberator tools implemented a set of static binary translation tools that worked without symbol table information most of the time [134]. However, that was mainly the fact because Honeywell designed their 200 series to be compatible with IBM's 1400 series [39]. Thus, only few additional tasks regarding incompatible instructions, except for direct instruction mapping were required to do.

However, as soon as a rewriter intends to add new instructions at instrumentation points the need for updating branch target addresses arises. While this is generally not a problem with program counter relative jumps and absolute branches, the update of target addresses for indirect branches is an undecidable problem in the general case [64]. A first attempt to tackle this problem has been made by Killian's Pixie in 1986 [130]. The tool implemented basic block counting by adding at least three MIPS 1 instructions of instrumentation code at the beginning of a newly discovered basic

block[20]. In order to cope for indirect branch target address updates, Killian introduced address translation tables. When using address translation tables, the rewriter adds a look-up table to the statically rewritten program that maps addresses from the original program into addresses of the modified program. Thus, every time an indirect branch is going to be taken in the modified program, an unconditional jump into the look-up table is performed, which redirects the program flow to a correct target address in the original application. Since the whole instruction address space of the original program must be covered and the original program must be stored as backup as well, this results in an overhead of more than 100 percent in terms of program size. Additionally, this approach is often limited to statically linked programs.

Today's address translation schemes implement an extended two-level version of Pxie's technique as it is used in Multiverse [17]. Here, a look-up table capable of addressing every byte in the application with a 4 byte offset is used to generate a mapping that allows reaching every entry in the table. Indexing an entry in the look-up table is done by calculating the offset of the original address from the base address of the original program's text section. If the calculation returns a valid address in the original text section it is returned. In case the default dummy value *0xffffffff* is returned, the program issues an error. However, if the indexed address is outside of the original text section, the second level look-up table is queried, which holds addresses to linked libraries.

In order to get rid of these high overheads, Sites et al. [122] implemented a so-called open-ended approach for static binary translation. Here, the translation is not complete after an initial run, but converges towards completeness upon new paths in the program are executed and translated.

Additionally, almost all of the early static binary rewriting tools realized their alteration using RISC processor architectures. One the one hand this imposes a relaxation on rewriting an instruction in-situ due to the fixed size instruction lengths of RISC processors. On the other hand, it raises challenges considering the rewriting of instruction having branch delay slots. For example, the "*branch equals*" instruction on MIPS features a delay slot, where its opposite instruction, "*branch not equals*" does not [35]. While this case in particular of interest for branch inversion in the context of optimization, it becomes a much more general problem when performing binary translation between different processor architectures like MIPS with some instructions having delay slots and ARM, where no delay slots are utilized [80]. Furthermore, when rewriting CISC programs, the variable instruction lengths have to be taken into account.

A static rewriting platform that supports x86, x86-64, SPARC, PowerPC, ARM architectures as well as a plethora of analysis algorithms is realized by Dyninst [48]. After its first years as a monolithic tool-set, it has been refactored into a set of modules directly exposing most of its Dyninst's features to the user in 2007 [109][21].

Rewriting with symbol table & relocation information. Rewriting with symbol table and relocation information at hand opens the possibility for more sophisticated static program alterations. Hence, a first idea implemented by Wall [140] was to reduce the high overhead introduced when using address translation tables by utilizing symbol and relocation information. The approach was realized before linking the binary together with an unmodified linker. The tool's focus was basic block counting.

Due to the availability of symbol and relocation information, static retrieval of control flow graphs that were augmented with additional information from prior profiling runs were possible. By using this idea, Larus et al. [80] implemented weighted control flow graphs that showed how often certain edges in the CFG were traversed. This made it possible to constrain that qualified for instrumentation and optimization.

[20] *load counter* → *increment counter* → *store counter*

[21] Publications building on Dyninst can be found at https://dyninst.org/related/view_papers.

Knowing which edges in a CFG are traversed often made rewriting with symbol and relocation information interesting for the application of link-time optimization. During the software building process the linker is invoked after code generation to create the final binary in its intended format (e.g., ELF). Thus, all required symbol information is still available in the object files, since stripping is done after linking. Code optimization at link-time is particular of interest since it enables program alterations that require knowledge of from an interprocedural CFG (ICFG)[22] that can be generated at link-time. Based on the ICFG several optimization approaches have been realized, with the most common being: **Constant propagation & subexpression elimination:** The basic idea behind constant propagation is to find pools of variables that do not change for a pair of abstract syntax tree nodes. This is of interest since the occurrence of these variables might be exchanged with their literal value, thus preserving memory references [70]. A similar idea is executed when by searching for sub expressions that can be pre calculated in order to save execution time [70]. Targeting those kind of rewritings have been identified as a promising source for performance improvements [116]. While earlier tools like Alto [93] and Plto [116] utilize straightforward constant propagation algorithms as presented in Wegman et al. [147], approaches that are more recent like Diablo implement fixed-point computation based solutions [29]. **Basic block rearrangement:** Tools that implement sophisticated profiling techniques such as Ispike [85] that targets Intel Itanium processors are able to obtain information that can be used for cache optimization and basic block rearrangement. In case of Ispike, the hotness[23] of a path is used to guide a basic block rearrangement algorithm in resorting certain blocks that are likely to be executed in a sequence. This is primarily done by inverting branches and inserting unconditional branches [85]. For more details on path hotness calculations see Section 4.2. In contrast, Diablo [29] implements a technique called factoring where identical basic blocks are merged and the edged are adapted accordingly. Both techniques are possible due to the availability of interprocedural CFGs, symbol tables and relocation information. **Unreachable code elimination:** Dead code is known as basic blocks that have no inbound edge and thus can safely be removed once the interprocedural CFG is created [93].

Additionally, prior to performing their tasks, link-time optimization tools use preliminary profiling runs to retrieve information about the locations that could benefit from optimization [29, 85, 93, 116].

4.2 Dynamic

Dynamic approaches either utilize the operating systems debugging or tracing interface like the PTRACE API under Linux [103] resp. the application debugging API under Windows [69], supply their own instrumentation runtime like Dynamo [11], or realize a virtual machine approach as done by STRATA [119].

One of the earliest dynamic binary rewriting approaches has been realized by Bergh et al. [20] in 1987. The tool performs binary translation from software running on HP-3000 machines to HP precision processors involving an emulator that is used whenever a data dependent or indirect branch must be resolved. In such a case a special node mapping table is constructed that is filled by the emulator translating HP precisions branches back to HP-3000 addresses Circumventing the problem of resolving indirect branches Johnson [68] realized a dynamic rewriter that utilized a modified linker, which retained all relocation information.

However, such direct approaches experience a significant overhead when instrumenting recurring instruction sequences. Furthermore, the required context switches between rewriting software

[22] An interprocedural CFG is a CFG that resembles all procedures of the program [5].

[23] Hotness is calculated by recording the recurrence of certain paths in a program. Since hot basic blocks cannot be isolated in general, path hotness is able to spread across basic block boundaries forming hot paths[136].

and program under investigation upon placing code at instrumentation code causes a significant run-time overhead [23, 117]. In order to reduce this overhead code caches are employed.

Code Caches. A code cache is a data structure that is filled with recurring instruction sequences that are subject to rewriting [11]. Cached instruction sequences are either of basic block [23, 35], or larger granularity (e.g. traces[24] [11, 127], or fragments [35, 117]). In case an instruction sequence supports larger granularity than basic blocks they usually span direct branches and end at indirect branches. Operating on instruction sequences that are larger than basic blocks has the advantage of being able to cache interprocedural instruction sequences. That directly benefits optimization attempts [11]. Nevertheless, approaches working with basic blocks employ basic block stitching techniques to enqueue several basic blocks to longer sequences [23, 35].

Selective Code Caching. However, not all sequences are necessarily cached, but only those that stem from *hot paths* during execution. A path gets hot once it has been traversed several times depending on a threshold value [11, 12, 35]. Nonetheless, during path hotness detection an instruction sequence, (e.g., a basic block) is not considered isolated, but is able to dissipate heat to its neighbor sequences. This is done by also taking the predecessor and successor sequences of a hot sequence into account. Therefore, the cache is only filled with heavily frequented parts of the program under investigation.

Usually caches employ some kind of replacement strategy in order to have always the most recently referenced information at hand. Considering code caches, the replacement strategy flush when full is most commonly used (e.g., [23, 35, 117]). An exception is implemented by Dynamo [11, 12] that realizes a pro-active cache flushing strategy based on how often cached traces have been hit recently. This results in a smaller code cache.

Instruction Sequence Linking. Adding instrumented instruction sequences into a code cache is first step on improving a dynamic rewriter performance. However, at the end of sequence a context switch between instrumented and instrumenting task would have to occur in order to find the start of the next sequence. Instruction sequence linking techniques counter this necessity.

Regarding unconditional and statically known conditional branches in Dynamo, fragments are linked together by letting a cached fragments exit branch point to its successor in the fragment cache rather than the original code location [11, 12]. Since also indirect and conditional branches are subject to target address redundancy, Dynamo maintains table of *predicted indirect branch targets* that is consulted upon reaching an indirect branch. If the indirect branch points at a cached fragment, the link is directly followed. Otherwise, control is handed back to the Dynamo runtime that resolves the new branch address and updates the entry in the *predicted indirect branch* table if it points to another already cached fragment.

Instruction sequence linking is also used by virtual machine approaches like STRATA [117]. Here, a fragment starts with the next uncached instruction and ends with a conditional or indirect branch. Since each fragment is placed into the cache, the according control transfer function is rewritten to reach in a trampoline that leads back to STRATA's fragment builder issuing a context switch. If, after some time both branch targets of a conditional branch reach into cached code, the trampolines are removed thus linking together the fragments. Something similar is used in terms of adding new fragments upon following the control transfer function is done when encountering indirect branches [118].

Another technique for indirect branch linking is utilized by Pin [84] and HDTrans [127]. Both tools implements a hash table that links the branch source addresses of indirect branches with a

[24] Although Dynamo names instruction sequences that qualify for caching traces, it calls them fragments as soon as they are in the code cache. [11] Hence we will call Dynamo's fragments *Dynamo fragments* throughout the paper.

list of predicted targets. A branch target address is added to the predicted target address list as soon as it has been taken once. This creates a linked list of indirect branch target addresses. The main difference between these two realizations is that HDTrans checks if the indirect branch target is already hashed and then links the basic blocks directly, while Pin resolves the target first, and then checks if it hit the right entry in the linked list of indirect branch targets associated with that originating address.

Cifuentes at al. [35] realized a simpler approach by counting the time a certain target is hit by an indirect branch, thus making it the default indirect branch target in their prediction. In case the prediction was false a context switch and resolution by the instrumenter was performed.

Usually return addresses can be resolved straightforward, however, traditional approaches required direct support for C++ exception handling, garbage collection and *longjmp()* calls. Thus, Sridhar et al. [127] introduced a cooperative call/return protocol that puts the untranslated return address on the stack and a translated return address in the code cache. Upon the call of a return instruction the translated address is taken. However, if the approach fails due to recursion for example, the indirect branch address translation scheme of HDTrans is used. Considering exception handling Chen et al. [30] instrument the appropriate *ntdll* calls in Windows to support this feature in Mojo.

Performing Alterations. After finding the instrumentation point of interest, dynamic rewriters either employ *(a)* minimal-invasive alteration to generate persistently changed binaries (e.g. [112]), *(b)* a full-translation approach allowing to execute elaborate analysis techniques (e.g. [53, 57]), or a *(c)* list of linked basic blocks that allow mostly for non-persistent binary alterations with less execution time overhead (e.g. [43, 75]).

In case *(a)*, the transformation problem falls back to the static minimal-invasive transformation problem after executing the analysis step dynamically (see Section 4.3). This has been done on in the work of Hollingsworth et al. [62] for Unix, and in Etch [112] for Microsoft's PE format. Minimal-invasive based constructs can be created at run-time by allocating a buffer holding the instrumentation code in an executable marked memory region using the *mprotect()*[25] function call under Linux or the *VirtualAllocEx()*[26] function under Windows. The same approach is used by DynInst [25, 48][27]. Diota [87] utilizes this approach as well, but retains an original copy of the altered program's parts in memory to handle the data in code and code in data rewriting problem for CISC machines.

Case *(b)* results in a static full-translation transformation problem. An instance is realized by the work of Jackson et al. and their rewriter Zipr [53] by implementing a lifter to the high-level language ADA[28] to utilize the plethora of available formal verification and analysis frameworks. Although a dynamic approach, Zipr claims to able to generate persistently rewritten binaries using a technique called address pinning (see Section 5 for further information). In order to facilitate transformation and optimization tasks when translating between IA-32 and IA-64 programs Srivastava et al. [49] employed Microsoft's own intermediate representation MSIL. Cifuentes et al [35] use a register transfer level based intermediate representation for their binary translator framework walkabout.

Case *(c)* subsumes rewriting approaches that implant their instrumentation directly at binary level using just in time compilation techniques. The basic approach does not require any trampolines or intermediate representations but injects compiled code fragments at the instrumentation points of the cached instruction sequences. Most of these approaches do not allow for persistent

[25] http://man7.org/linux/man-pages/man2/mprotect.2.html
[26] https://msdn.microsoft.com/en-us/library/windows/desktop/aa366890(v=vs.85).aspx
[27] See https://www.dyninst.org/related/view_papers for a list of publications using DynInst.
[28] http://www.adaic.org/

transformations, but rather aim at efficient run-time alterations during program execution as it is done by Kiriansky et al. [75], Pin[29] [84] and the dynamic path of Kevlar [43] for example. Since not all alterations[30] require a detailed disassembly of each instruction, DynamoRIO[31] [23] introduces a five layer based instruction representation scheme. The scheme is an extension to Dynamo's [11, 12] instruction representation that is used for fragment linking.

4.3 Minimal-invasive

Minimal-invasive binary transformation in a static rewriting attempt works as depicted in Fig. 4a to Fig. 4c. The program flow displayed in Fig. 4a will be redirected during the transformation procedure shown in Fig. 4b in such a way that the intended instrumentation code located in a new program section (respectively segment) will be executed before and after the instrumented function in Fig. 4c.

It has to be noted that minimal-invasive binary transformation attempts are only capable of instrumenting binaries at branch granularity, since instrumentation code can only be reached by redirecting existing branches in the original binary.

The original program flow displayed in Fig. 4a *(1)* through *(4)* consists of a prologue and epilogue for function *a_func* with *a_func* performing an arbitrary task. It will be augmented by some instrumentation code adding features to *a_func*s epilogue and prologue code as it would be the case when adding stack canaries[32] for security. *a_func*s transformation process displayed in Fig. 4b consists of the following steps, assuming the binary format under investigation is of type ELF:

(I) A new section (*.newsec*) is created and the instrumentation code is added to the binary.

(II) The function prologue of *a_func* is copied right after the instrumentation code. Furthermore, the instrumentation code has to save the current machine state in order to stay transparent for the remainder instructions and the instrumented binary. The old location of *a_func*'s prologue is called the instrumentation point. It must be long enough to hold the instructions added in step (III). The instrumentation code itself is shown as *exec_instrum* comment. Furthermore, an unconditional jump is added ("`jmp <ra_func>`") aiming right before the call instruction for *a_func*

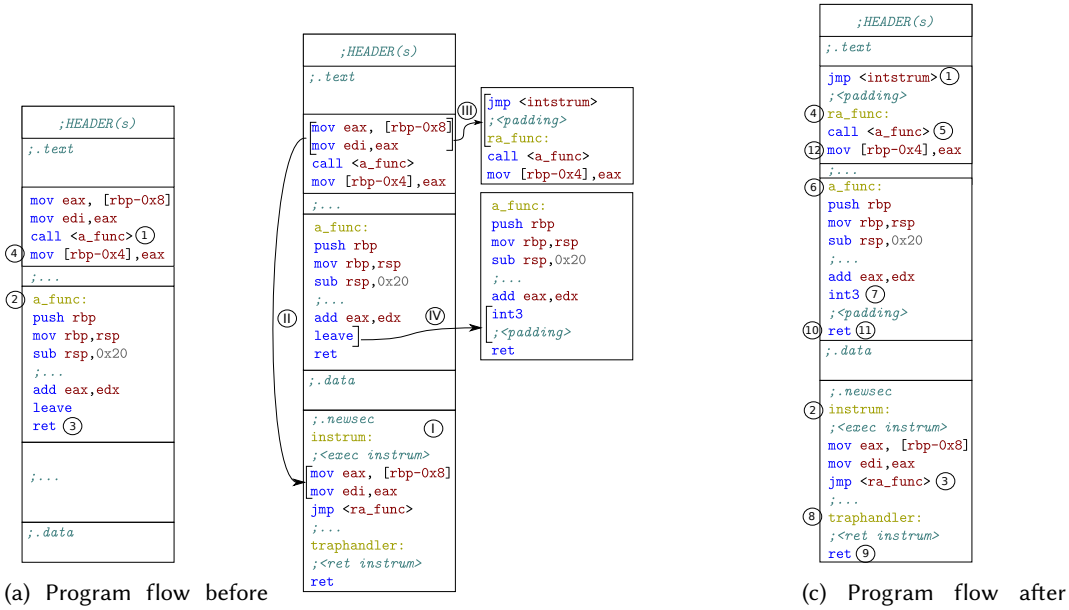
(III) There is a check if the prologue of *a_func* can be exchanged with an unconditional jump into the instrumentation code located at *instrum.* in *.newsec*. While this was possible in our case, not all instruction set architectures are able to cover their whole address space within in a single jump instruction [131]. Thus, when the instrumentation code location is out of range for a single jump instruction another instruction would have to be inserted in order to reach *instrum.* in *.newsec*. This in turn would cause address updates of possibly all branches within the binary, leading to the necessity of reassembling the whole binary, which is not intended, or even possible in most cases when applying minimal-invasive binary rewriting [46]. A solution to this problem was presented by Prasad and Chiueh [106] who replace the jump instruction with a debug interrupt call (if available on the architecture). In case explicit debug instructions are not available or their utilization is not intended, trampolines can be used [145]. Similar to a real trampoline, a trampoline in the context of binary rewriting can be reached with a comparably small jump from an instrumentation point, however, just like

[29]Pin for ARM adds a relaxation to the rewriting problem when it comes to self modifying code detection, as ARM processors provide a dedicated instruction for that purpose [60].

[30]E.g., Detection of basic block boundaries requires only to know that a certain byte sequence represents a branch instruction, but the branch's parameter decoding is not of interest at that time.

[31]A list of publications utilizing DynamoRio can be found at http://dynamorio.org/pubs.html

[32]A random, but known by the application value that is placed before the return instruction of a function. In case it is overwritten by an adversary, or by accident, the program will know and abort before any harm is done.



(a) Program flow before transformation.

(b) Transformation steps.

(c) Program flow after transformation.

Fig. 4. Implications of (static) minimal-invasive binary transformation on a binaries program flow. Round circles with Arabic numbers unveil the program's branch sources (right hand side of the corresponding instruction) and branch targets (left hand side of the corresponding instruction) shown in Fig. 4a and Fig. 4c. In addition, round circles with Roman numbers reflect the necessary transformation steps in order of their application as displayed in Fig. 4b.

when using a real trampoline, the subsequent jump will be amplified making it possible to reach a far away instrumentation point of interest in a different section. The trampoline is usually implemented as a jump table, with the instrumentation points addressing a slot within the table to reach their intended instrumentation code. In case trampolines and debug instructions are not an option, one may use the instrumentation point and its adjacent instruction to create a branch to instrumentation code as it is done by BIRD [96].

(IV) Finally, the epilogue of *a_func* is instrumented. However, the "`leave`" instruction occupies too little memory to be replaced by a "`jmp`" instruction. This also represents a case in which Prasad and Chiueh's [106] solution of adding a debug interrupt call can be applied as shown in Fig. 4b.

The program's transformed flow is displayed in Fig. 4c. Here, the first jump instruction *(1)* targets the instrumentation code *(2)*. After prologue instrumentation has been reached, the context is restored and an unconditional branch back before the call of *a_func* is performed in *(3)* through *(5)*. *a_func* is executed until the epilogue instrumentation is called by the debug interrupt instruction in step *(7)*. Debug interrupts are often implemented as software interrupts in general purpose operating systems. They essentially cause a signal to be raised by the operating system kernel; therefore, a corresponding signal handler can be registered that is executed in turn in step *(8)*. Finally, in steps *(9)* through *(11)* the signal handler returns to *a_func*, which eventually returns to step *(12)* and thus leaving the instrumentation detour.

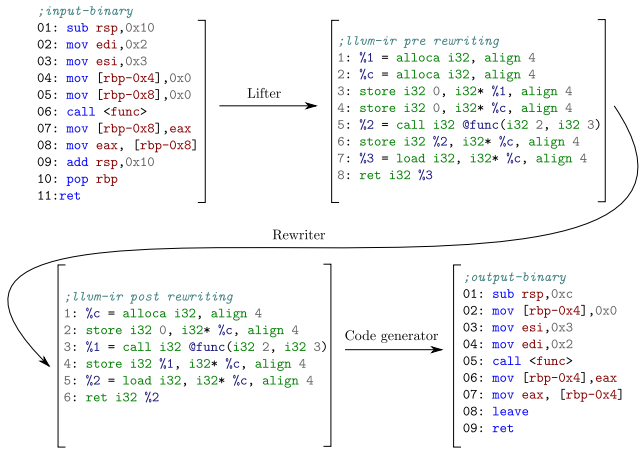


Fig. 5. Full-translation binary transformation in a stack usage optimization scenario using the LLVM-IR as intermediate representation. The reassembly step (Code generator) is added to show the impact of its individual decisions on the final binary output leading to a different code layout.

4.4 Full-translation

Full-translation allows for binary alteration in instruction granularity. Hence, every instruction is a possible instrumentation point.

Fig. 5 shows the general approach on a stack usage optimization scenario. The goal is to reduce the stack memory consumption of the example function. First, the *input-binary*, as a direct result of the disassembler, is fed into a lifting algorithm, which maps every instruction to a semantically equivalent intermediate representation statement [6, 31]. In our case, the LLVM [83] intermediate representation (IR) is used. The local variable created in line *4* of the *input-binary* block displayed in Fig. 5 is never used throughout the function; therefore, it is a candidate for removal to save stack space. Therefore, the IR in block *llvm-ir pre transformation* is altered in such a way that only a single local variable is used in the *llvm-ir post transformation*. Binary transformation is performed on the IR, which in turn is reassembled using a code generator. Since code generated by different compilers (e.g., gcc, clang) as well as compiler versions (e.g., gcc 3 vs. gcc 5) might make different choices regarding what IR instruction is mapped to an actual instruction; the *output-binary* might have a different address layout and caching behavior [6]. In our example, the function prologue in *input-binary* is changed from "`add rsp, 0x10`" and "`pop rbp`" to "`leave`" by the code generator. Note that both epilogues are semantically equivalent but lead to a different memory layout of the transformed binary.

Another source of noise for the semantically equivalent lifting approach is the use of advanced processor instructions that must be directly supported by the IR, such as the multiply accumulate instruction on an ARMv7 architecture. The multiply accumulate instruction executes the multiplication and addition (i.e., $d = (a \times b) + c$) of three distinct values that are stored at a possibly distinct target register. It is often used for recurring, but speed oriented operations such as image filtering. Depending on the lifter's features the instruction can either be lifted into a semantically equivalent, or instruction equivalent IR representation. In case an instruction equivalent lifter is used, the code generator will emit another MLA instruction after transformation. When using a semantically equivalent lifter this might not be the case. Although an optimizing code generator can detect the MLA pattern in the IR, it might be the case that an instrumentation point is located

after the location of the former MLA instruction. The transformer will add its instrumentation code, thus preventing an optimization algorithm from detecting the MLA pattern. This will result in a binary with different memory layout and different execution speed due to the usage of the less efficient distinct multiply and addition instructions.

Furthermore, when inserting a set of instructions between two operations relying on the integrity of the flags register, the rewriter must either choose *(a)* to use instructions not affecting the flags register, *(b)* safe the flags register to a spare register if available, or *(c)* save the flags register onto the stack. Nevertheless, as stack save and restore operations imply additional two instructions this is only feasible if larger blocks of instructions are inserted. Additionally, in case of using CISC machines, spare registers are rare [97].

OM by Srivastava and Wall [5] from 1992 implemented a link-time code manipulation system that used register transfer level representation to manipulate the program under investigation. The tool used weighted graphs[33] for loop detection in order to move loop invariant before the loop's body.

One of the first full-translation based binary rewriters that did not require symbol information was implemented by Larus and Schnarr [81]. Their tool EEL is based on ATOM and lifts binaries into an unspecified intermediate representation. The approach implemented a callback to instrumentation code that is not located at a distinct section as it is done in ATOM using a minimal-invasive approach. Furthermore, unrecognized parts of the binary during disassembly and lifting are stored as backup in the altered binary producing a high overhead. High space overhead due to keeping backup copies because of imperfect structural recovery algorithms is a central issue in full-translation approaches.

The first work tackling this issue was done by Wang et al. with Uroboros [142] in 2015 and Ramblr [141] in 2017. A different idea was developed by Hasabnis and Sekar [55]. The authors' approach is to utilize machine learning in order to obtain completely and correctly lifted binaries into gcc's internal representation RTL. A training data set is generated by tracing the compilers RTL to assembly code decisions for a large set of compilations of different programs. Their framework does elastic pattern matching to map recovered instructions and their parameters back into the gcc intermediate representation RTL when operating on different binaries than those the tool has been trained with.

Since intermediate representations (IR) are the central component in full-translation binary transformation, the following paragraphs provide a compact overview on available IRs that are used throughout the tools mentioned in our paper.

Intermediate representations: The following list presents the most common intermediate representations (IRs) together with their availability and properties for the use case of binary rewriting.

REIL [47] is used as IR in IDA-PRO [114] and implements a reduced instruction set of x86, PowerPC and ARM processors. The reduction results in the most common instruction responsible for security related bugs. In order to use REIL without IDA, an open source implementation of REIL, called OpenReil [101] which supports x86 and parts of the ARMv7 architecture including thumb mode. Currently, 23 instruction are supported by OpenREIL. **BAP** [24] is a complete binary analysis framework implementing the intermediate representation BAP. At the time of writing the BAP IR supports x86, ARM, MIPS and PowerPC CPUs. It was designed to represent all side-effects (flags, wait cycles, etc.) to enable syntax directed analysis. The semantics have been formally verified to aid in formal reasoning on program properties. The intermediate representation of **LLVM** [83] provides a complete representation of all supported instructions. Unfortunately, it lacks a direct lifting support, which is thus provided by the lifting framework MCSema from trail of bits [100]. The Tiny Code Generator **(TCG)** IR used in QEMU supports a wide variety of processor architectures.

[33] Weighted graphs reflect the recurrence of basic blocks [5].

Nevertheless, it is strongly intertwined with QEMU itself [107], making it hard to use it outside the project. Fortunately, there exists an extracted version of TCG as a library currently supporting a subset of TCGs features [153]. **VEX** [138] is the intermediate representation of Valgrind, thus supporting a wide variety of common processor architectures as well as a subset of specialized instructions such as x86 MMX. It is successfully used in the angr [121] binary analysis framework. An exception from the traditional approaches is **LISC**. LISC learns instructions semantics to form an IR from compiled binaries [56], thus achieving a higher instruction coverage rate. However, its success varies greatly with the diversity of the supplied test data.

5 CODE GENERATION

The code generation step integrates the changes made during the transformation step into an executable binary. This can be done temporarily in a non-persistent (dynamic) binary rewriting scheme, or durably in a persistent approach. While a temporary rewriting attempt only has to make the instrumentation code reachable during execution *(A)*, a persistent solution must either integrate the instrumentation code persistently into an existing binary *(B)* and alter its administrative components, or generate a completely new binary from scratch *(C)*.

(A) Making instrumentation code reachable during execution: At the end of the transformation step in Section 4.2, step *(c)*, the intended instrumentation code is ready to be inserted in the program flow at the instrumentation point. First, the instrumentation code is fed to a just-in-time (JIT) assembler (see [23]). Next, the binary snippet is placed into a newly allocated buffer that is configured to have execution permissions[34]. Finally, the branch target of the instrumentation point is altered making the new code reachable during execution.

(B) Integrating instrumentation code persistently into an existing binary: This approach causes some overhead in terms of disk size depending on the used transformation technique covered in Section 4. If a minimal-invasive algorithm has been employed, the instrumentation snippet is fed into an assembler and added into a new section within the altered binary. Moreover, the VMA (virtual memory address) of the newly added section must be associated with an executable segment when running Unix based operating systems. This can be achieved by using the binutils bfd [129] and elf [132] library. In case of Windows, the newly created section must be registered at the section table [105].

In case a full-translation approach is utilized, the binary under investigation must be fully recreated. However, due to the challenges regarding the analysis steps (e.g., the indirect branch resolution problem, parts of the binary might not have been identified as code or data at all. Since these parts are likely to be vital for the proper function of the binary, they still have to be integrated in the rewritten binary. This can be done by interpreting an unidentified binary stream as both code and data and put it in both sections (*.data*, *.text*) during code generation. Although, this possibly results in a significant overhead, it resolves the indirect branch resolution problem, as all variables occurring in the unidentified binary blob can be referenced in its data section copy. Furthermore, all indirect branches can be resolved to the blob's text section copy. Nevertheless, the code generation part must be executed by an altered code generator that has knowledge about the original binary's layout [102]. This is mainly caused by the fact that binaries have no concept of symbols and labels [142] which are required by a commercially available off-the-shelf assembler to create a fully functional binary from source code.

[34]Today, the execution of code stemming from the data section of a program is often prohibited for security reasons. However, it is possible to set certain pages that previously were set to non executable to executable again

(C) Generating a completely new binary off the shelf: Throwing away the old binary file and creating a completely new one after the transformation step is a method to keep the additional overhead small (e.g. backup copies of unidentified raw binary snippets). The success of this approach is directly related to the success of the recovery strategies executed during the analysis steps. The so-called reassembling procedure itself is direct [141]. First, labels are distributed to all recovered symbols and symbolizable immediates. Then the augmented disassembly can be put into a single file and fed to an commercially off-the-shelf disassembler such as nasm[35].

6 CATEGORIZATION

So far, we have covered all main steps in performing binary writing, their problems and the techniques attempting to solve them. What is still missing is a mapping between the main application domains, the basic techniques and the tools implementing these solutions (see Table 1 and Fig. 6). The 67 publications present an overview on the available tools and papers concerning either persistent or non-persistent binary rewriting, covering a period from 1966 to 2018.

The items in the pool of publications are sorted with respect to their year of publication and categorized according to their overall use case-*Emulation*, *Optimization*, *Observation*, or *Hardening*. Furthermore, the category *Generic* was added to reflect solutions that have been implemented with a versatile usage in mind.

Table 1 displays the architecture of the respective transformation procedure, the operating system the tool is executed on, the applied structural recovery mechanisms and the realized level of persistence. The resulting categorization allows us to provide a comprehensive overview with respect to the most distinguishable features the investigated publications offer.

Table 1. List of publications under comparison with the column *Tool/Author* showing either the respective tool's name, or the publication authors in the form "{first paper author} et al.". Adjacent, column *Structural recovery* providing additional information on the structural recovery techniques used. Column *Aim* shows the tools initial application scenario as introduced in Section 1. Besides the terms *Emulation*, *Observation*, *Optimization* and *Hardening*, the category *Generic* was added to indicate the tool's intended versatile usability. Finally, column *Pers.* shows if the respective paper allows for persistent binary rewriting.

#	Tool/*Author*	Ref.	Year	Arch.	System	Structural recovery	Aim	Pers.
1	Liberator	[63]	1966	IBM1400/Honeywell 200	honeywell os	direct translation due to ISA compatibility	Emulation	yes
2	Pixie	[130]	1986	risc	unix	address translation table (indirect jumps)	Optimization	yes
3	*Bergh et al.*	[20]	1987	HP3000	unix	symbol table	Emulation	yes
4	Mimic	[89]	1987	cisc	system/370 os's	run-time	Emulation	yes
5	*Bedichek et al.*	[18]	1990	motorola 88000	vunix	run-time	Emulation	no
6	*Johnson et al.*	[68]	1990	risc	unix	symbol table	Optimization	yes
7	Accelerator	[7]	1992	risc	unix	run-time as fallback	Emulation	yes
8	OM	[5]	1992	risc	unix	symbol table	Optimization	yes
9	*Hollingsworth et al.*	[62]	1994	risc	unix	run-time	Optimization	no
10	Shade	[36]	1995	sparcv8	sparcv8	run-time	Emulation	no
11	*Wahbe et al.*	[54]	1995	mips, sparc	ultrix	symbol table	Emulation	yes
12	TIBBIT	[37]	1995	motorola 68000	ibm rs6000, aix3.2	run-time	Emulation	yes
13	EEL	[81]	1995	x86	windows	pattern matching, backward slicing (indirect jumps)	Generic	yes
14	ATOM	[128]	1997	risc	unix	symbol table	Generic	yes
15	Etch	[112]	1997	x86	windows	unspecified	Optimization	yes
16	Kerninstd	[131]	1999	solaris	solaris	run-time	Generic	no
17	DynInst I	[25]	2000	x86-64, powerpc, armv8	windows, linux	run-time	Optimization	no
18	UQBT	[33]	2000	x86, sparc, jvm	windows, linux	pattern	Emulation	yes
19	Vulcan	[49]	2001	x86	windows	run-time/pattern matching	Optimization	yes
20	*Kiriansky et al.*	[75]	2002	solaris	solaris	run-time	Hardening	no
21	STRATA	[119]	2003	x86, mips, sparc	linux, unix	run-time	Emulation	no
22	*Prasad et al.*	[106]	2003	x86	linux	pattern matching	Hardening	yes

Continued on next page

[35] http://www.nasm.us/

Table 1 – continued from previous page

#	Tool/*Author*	Ref.	Year	Arch.	System	Structural recovery	Aim	Pers.
23	DynamoRIO	[23]	2003	x86	windows, linux	run-time	Generic	no
24	Diablo	[45]	2004	arm	linux	heuristics	Optimization	yes
25	Pin	[84]	2005	x86-64, arm, itanium	linux	run-time	Generic	no
26	*Ligatti et al.*	[1]	2005	x86	windows	run-time/pattern matching	Hardening	yes
27	LANCET	[139]	2005	arm	linux	heuristics	Optimization	yes
28	*Hu et al.*	[65]	2006	x86, mips, sparc	linux, unix	run-time	Hardening	no
29	BIRD	[96]	2006	x86	windows	pattern matching	Hardening	both
30	PittSFIeld	[90]	2006	x86	linux	pattern matching, forced instruction alignment	Hardening	yes
31	Valgrind	[98]	2007	arm, mips, x86, ppc32, s390	linux, unix	run-time	Observation	no
32	BitBlaze	[126]	2008	x86, arm	linux	VSA (indirect jumps), SSA analysis	Hardening	no
33	Jakstab	[72]	2008	x86	windows	abstract interpretation, bounded address tracking (pointer type/integer distinction)	Generic	no
34	Pebil	[82]	2010	x86	linux	pattern matching	Observation	yes
35	SecondWrite	[102]	2011	x86	linux	speculation	Hardening	yes
36	Howard	[123]	2011	x86	linux	VSA, abstract interpretation, abstract data structure identification, (bounded) pointer tracking	Observation	yes
37	ROPdefender	[42]	2011	x86	linux	run-time	Hardening	no
38	BAP	[24]	2011	x86, arm	linux	VSA (indirect jumps), supports for ist analysis SSA	Observation	no
39	STIR	[144]	2012	x86	windows, linux	pattern matching, heuristics (from ida)	Hardening	yes
40	REINS	[145]	2012	x86	windows, linux	pattern matching, heuristics (from ida)	Hardening	yes
41	BinArmor	[124]	2012	x86	linux	VSA, abstract interpretation, abstract data structure identification, (bounded) pointer tracking	Hardening	yes
42	CFFIR	[156]	2013	x86	windows	pattern matching	Hardening	yes
43	SLX	[111]	2013	x86	linux	run-time	Hardening	no
44	*Anand et al.*	[6]	2013	x86	linux	speculation, dynamic exec. (indirect jumps), SVA	Generic	yes
45	MADRAS	[137]	2013	x86-64	linux	pattern matching	Generic	yes
46	*Smithson et al.*	[125]	2013	x86	linux	speculation, added dynamic exec. (indirect jumps)	Generic	yes
47	FPGate	[155]	2013	x86	linux	pattern matching	Hardening	yes
48	*Zhang et al.*	[158]	2013	x86	linux	pattern matching	Hardening	yes
49	P$\tilde{S}$I	[157]	2014	x86	linux	pattern matching	Hardening	yes
50	*Davidson et al.*	[43]	2015	x86, mips, sparc	linux	run-time	Hardening	no
51	VTInt	[154]	2015	x86	linux	pattern matching	Hardening	yes
52	Lockdown	[104]	2015	x86	linux	run-time, pattern matching, data section checks to find static pointers to instructions	Hardening	no
53	UROBOROS	[142]	2015	valgrind	linux, unix	backward slicing, VSA, heuristics, symbolic execution	Generic	yes
54	*Davidson et al.*	[44]	2016	x86	linux	run-time	Hardening	no
55	Kevlar	[53]	2016	x86	linux	run-time	Generic	yes
56	*Hawkins et al.*	[58]	2016	x86-64	linux	run-time	Hardening	yes
57	Dyninst II	[48]	2016	x86-64, ppc64, sparc, arm64	linux, unix, windows	run-time, pattern matching, machine-learning	Generic	both
58	*Wang et al.*	[143]	2017	x86	QEMU	run-time	Emulation	no
59	Zipr	[57]	2017	x86-64	linux	run-time	Hardening	yes
60	rev. ng	[46]	2017	QEMU	QEMU	simple expression tracking, offset range data flow analysis	Hardening	yes
61	Ramblr	[141]	2017	QEMU	QEMU	backward slicing, VSA, heuristics, symbolic execution	Observation	yes
62	CFI CaRE	[99]	2017	armv8	armv8	pattern matching on function epilogues and prologues	Hardening	yes
63	RevARM	[71]	2017	arm	arm	pattern-matching, backward slicing (indirect jumps)	Hardening	yes
64	QDIME	[10]	2017	x86-64, arm, itanium	linux	run-time	Observation	no
65	RL-Bin	[88]	2017	x86	linux	speculation, added dynamic exec. (indirect jumps)	Generic	yes
66	Zipr++	[61]	2017	x86-64	linux	run-time	Hardening	yes
67	Multiverse	[17]	2018	x86	linux	iterative linear sweep	Generic	yes

In order to provide a better insight in each tool's entanglement with the used disassembly approach, structural recovery mechanism, and transformation approach, Fig. 6 depicts a Sankey diagram starting from the uncategorized pool of 67 publications on the left-hand side. We show the first categorization step into the tools' primary application domain, the disassembly algorithm of choice, the selected structural recovery mechanism, and the utilized transformation schema. The numbers shown in the Sankey diagram reflect the publications' index number in Table 1.

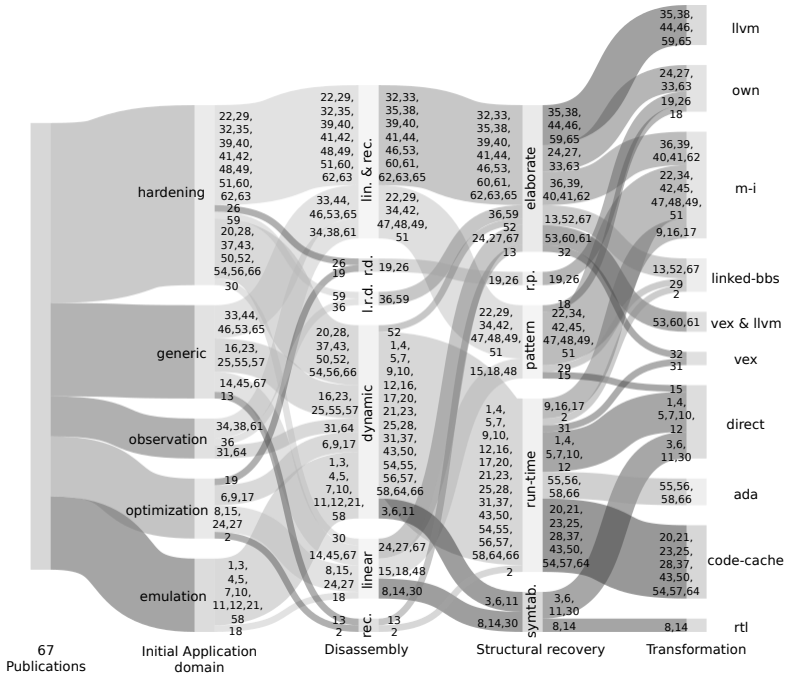


Fig. 6. Sankey diagram further categorizing the publications listed in Table 1, starting from an unsorted pool of 67 publications. The adjacent row depicts the tool's application domain, followed by its used disassembly, structural recovery and transformation strategy. Each number displayed in this figure represents the index number in Table 1. For example, the publication with the index number 11 in Table 1 residing in the *Emulation* domain implements a dynamic disassembly approach together with a symbol table based algorithm for structural recovery and implements a direct transformation scheme. Regarding the three columns *Disassembly*, *Structural recovery* and *Transformation*, the adjacent abbreviations have the following meaning: lin. → linear sweep; rec. → recursive traversal; r.d. → recursive traversal & dynamic; l.r.d. → linear sweep & recursive traversal & dynamic; dyn. → dynamic; symtab. → symbol table; m-i → minimal-invasive; linked-bbs → linked list of basic blocks. Furthermore, the *direct* transformation scheme references rewriting approaches that directly alter the instructions of interest without any kind of administrative structures. In case the approach implements a full-translation transformation scheme the utilized intermediate representation is displayed instead of the term *full-translation* to further detail the comparison.

Furthermore, as can be seen in Fig. 6, besides 29 dynamic disassembly approaches, 23 publications implement mixed linear-sweep/recursive traversal static disassemblers, forming the two most popular disassembly approaches. Considering the largest group of the remaining publications, linear-only disassembly is implemented by 9 tools.

In the structural recovery step, 27 out of 29 tools implementing dynamic disassembly utilize runtime structural recovery. The 34 tools using the mixed static disassembly approach split into structural recovery approaches utilizing pattern matching (11), elaborate (19), symbol table based (3), and run-time (1) based techniques. Pattern matching based structural recovery furthermore includes the remaining disassembly ideas.

The fourth column of Fig. 6 lists 24 publications that show a large diversification of full-translation approaches consisting of LLVM, own IRs, VEX, gcc register transfer level (RTL) and ADA (the programming language) as well as VEX and LLVM hybrids. The second largest group is constituted

by minimal-invasive rewriting tools (16) followed by code-cache based tools (11) and rewriters employing direct strategies (11). Additionally, 46 out of 67 publications implement a persistent binary transformation scheme, with BIRD [96] and Dyninst [48] being able to do both as shown in Table 1.

7 CONCLUSION

In this survey, we provided an in-depth overview on the development and the state of the art in binary rewriting. Our work covers a time period from 1966 to 2018, beginning with one of the first rewriting programs like Honeywell's [63] dynamic translation utilities and the static instrumenter Pixie [130], to tools like Ramblr [141] that are able to use a standard assembler for code generation, or the superset disassembly based approach implemented by Multiverse [17]. We addressed the necessary steps for binary rewriting in detail, namely *disassembly*, *structural recovery*, *transformation* and *code generation*.

In our analysis, we categorized 67 publications directly related with binary rewriting tools within the identified areas of application (*emulation*, *observation*, *optimization hardening*, and *generic*) with respect to used techniques in each of the steps mentioned above. We discussed their particular challenges as well as their solutions, which further revealed the following incomplete list of open research challenges.

Analysis. Regarding the disassembly step, for CISC architectures the detection of instruction boundaries is still an open problem. Furthermore, the reliable distinction between data and code is of concern. However, this problem is undecidable in the general case. Sound and precise CFG recovery is believed to be undecidable in the general case [27, 51]. Nevertheless, considering function recovery, graph based methods as used by Federico and Agosto. [52] and Federico et al. [46] offer promising results, but still lack precision in the advent of aggressive compiler optimization. While resolving indirect branches is an undecidable problem in the general case, Bauman et al. [17] showed that an iterative linear sweep approach in combination with a look up table based attempt and a dynamic indirect branch resolution scheme is a viable approach to solve this problem. However, this approach has a high overhead in terms of static memory consumption. The overhead amounts to 4 times the addressable text section size of the binary under investigation when targeting IA-32 based systems. Hence, another identified challenge is to decrease the memory overhead in static indirect branch prediction schemes.

Transformation. While full-translation based rewriting schemes allow for the application of various reasoning approaches due to the more abstract representation of the binary under investigation, currently only semantic equivalent lifters are available. Regarding their maturity, we discovered that only those intermediate representations with a large backing community such as *VEX* and *LLVM-IR* provide the sophistication to support more than a set of core instructions. Although, semantic equivalent lifting is sufficient for many applications, scenarios like altering timing sensitive applications, performance optimization for throughput-oriented programs, or rewriting software with real-time requirements would greatly benefit from instruction equivalent lifters. While the realization of instruction equivalent lifting seems more like an engineering challenge to us, we believe that the investigation of the additional application scenarios such an addition would make fall into the category of research challenges. Furthermore, additional extensions to approaches like Uroboros [142] and Ramblr [141] would aid in lowering the overhead introduced by static binary rewriting attempts.

While today, the x86 architecture is still the primary target for binary rewriting applications for reasons like complexity (*If we did it here, we can do it everywhere.*) and availability, other architectures like ARM and MIPS, and their particular characteristics draw more and more interest.

This might be a challenge especially for low- to mid-end hardware platforms that are employed in Internet-of-Things environments.

ACKNOWLEDGEMENTS

This work was partly supported by the City of Vienna, MA 23 under the project grant number MA 23-Project 17-06. This work was supported partly by the Christian Doppler Forschungsgesellschaft (CDG) through the Josef Ressel Center (JRC) project TARGET and the Austrian Research Promotion Agency (FFG) through the project SBA-K1 (854188). The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS '05)*. ACM, New York, NY, USA, 340–353.

[2] Hiralal Agrawal. 1994. On Slicing Programs with Jump Statements. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94)*. ACM, New York, NY, USA, 302–312.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, Principles, Techniques. *Addison Wesley* 7, 8 (1986), 9.

[4] Frances E. Allen. 1970. Control Flow Analysis. *SIGPLAN Not.* 5, 7 (July 1970), 1–19.

[5] David W. Wall Amitabh Srivastava. 1992. *A Practical System for Intermodule Code Optimization at Link-Time*. Technical Report. Digital Western Research Laboratory.

[6] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 295–308.

[7] Kristy Andrews and Duane Sand. 1992. Migrating a CISC computer family onto RISC via object code translation. In *ACM Sigplan Notices*, Vol. 27. ACM, 213–222.

[8] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Security Symposium*.

[9] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *IEEE European Symposium on Security and Privacy*.

[10] P. Arafa, G. M. Tchamgoue, H. Kashif, and S. Fischmeister. 2017. QDIME: QoS-Aware Dynamic Binary Instrumentation. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 132–142.

[11] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 1999. *Transparent dynamic optimization: The design and implementation of Dynamo*. Technical Report. Hewlett-Packard.

[12] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2011. Dynamo: a transparent dynamic optimization system. *Acm Sigplan Notices* 46, 4 (2011), 41–52.

[13] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *Compiler Construction*. Springer, 2732–2733.

[14] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages.

[15] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3, Article 50 (May 2018), 39 pages.

[16] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to recognize functions in binary code. USENIX.

[17] Erick Bauman, Zhiqiang Lin, and Kevin Hamlen. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*. San Diego, CA.

[18] Robert Bedichek. 1990. Some efficient architecture simulation techniques. In *Proceedings of the Winter 1990 USENIX Conference*. 53–64.

[19] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*. 41–46. https://www.usenix.org/legacy/event/usenix05/tech/freenix/full_papers/bellard/bellard_html/

[20] Arndt B Bergh, Keith Keilman, Daniel J Magenheimer, and James A Miller. 1987. Hp-3000 emulation on hp precision architecture computers. *Hewlett-Packard Journal* 38, 11 (1987), 87–89.

[21] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE '11)*. ACM, New York, NY, USA, 9–16.
[22] David W Binkley and Keith Brian Gallagher. 1996. Program slicing. *Advances in Computers* 43 (1996), 1–50.
[23] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*. IEEE, 265–275.
[24] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. *BAP: A Binary Analysis Platform*. Springer Berlin Heidelberg, Berlin, Heidelberg, 463–469.
[25] Bryan Buck and Jeffrey K Hollingsworth. 2000. An API for runtime code patching. *The International Journal of High Performance Computing Applications* 14, 4 (2000), 317–329.
[26] Juan Caballero and Zhiqiang Lin. 2016. Type Inference on Executables. *ACM Comput. Surv.* 48, 4, Article 65 (May 2016), 35 pages.
[27] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. 2015. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium*. 161–176.
[28] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. 2012. Unleashing Mayhem on Binary Code. In *2012 IEEE Symposium on Security and Privacy*. 380–394.
[29] D. Chanet, B. De Bus, B. De Sutter, L. Van Put, and K. De Bosschere. 2005. DIABLO: a reliable, retargetable and extensible link-time rewriting framework. In *2005 IEEE International Symposium on Signal Processing and Information Technology(ISSPIT)*, Vol. 00. 7–12.
[30] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M Gillies. 2000. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*. 81–90.
[31] David Chisnall. 2013. The challenge of cross-language interoperability. *Commun. ACM* 56, 12 (2013), 50–56.
[32] Cifuentes and Malhotra. 1996. Binary translation: static, dynamic, retargetable?. In *1996 Proceedings of International Conference on Software Maintenance*. 340–349.
[33] C. Cifuentes and M. Van Emmerik. 2000. UQBT: adaptable binary translation at low cost. *Computer* 33, 3 (Mar 2000), 60–66.
[34] C. Cifuentes and A. Fraboulet. 1997. Intraprocedural static slicing of binary executables. In *1997 Proceedings International Conference on Software Maintenance*. 188–195.
[35] Cristina Cifuentes, Brian Lewis, and David Ung. 2002. *Walkabout: A retargetable dynamic binary translation framework*. Technical Report. Sun Microsystems, Inc.
[36] Bob Cmelik and David Keppel. 1995. Shade: A fast instruction-set simulator for execution profiling. In *Fast Simulation of Computer Architectures*. Springer, 5–46.
[37] Bryce H Cogswell and Z Segall. 1995. Timing insensitive binary-to-binary migration across multiprocessor architectures. In *wpdrts*. IEEE, 193.
[38] Lucian Cojocar, Taddeus Kroes, and Herbert Bos. 2017. *JTR: A Binary Solution for Switch-Case Recovery*. Springer International Publishing, Cham, 177–195.
[39] DATAPRO RESEARCH CORPORATION. 1974. Honeywell Series 200 and 2000. www.bitsavers.org/pdf/honeywell/datapro/70C-480-01_7404_Honeywell_200_2000.pdf
[40] Patrick Cousot and Radhia Cousot. 1992. Abstract interpretation frameworks. *Journal of logic and computation* 2, 4 (1992), 511–547.
[41] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. 1998. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, Vol. 98. San Antonio, TX, 63–78.
[42] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. 2011. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM, 40–51.
[43] J. W. Davidson, J. Hiser, A. Nguyen-Tuong, M. Co, B. D. Rodes, and J. C. Knight. 2015. Security protection of binary programs. In *10th IET System Safety and Cyber-Security Conference 2015*. 1–6.
[44] J. W. Davidson, J. D. Hiser, A. Nguyen-Tuong, C. L. Coleman, W. H. Hawkins, J. C. Knight, B. D. Rodes, and A. B. Hocking. 2016. A System for the Security Protection of Embedded Binary Programs. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. 234–237.
[45] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. 2004. Link-time Optimization of ARM Binaries. *SIGPLAN Not.* 39, 7 (June 2004), 211–220.
[46] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev. ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*. ACM, 131–141.

[47] Thomas Dullien and Sebastian Porst. 2009. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. http://www.zynamics.com/downloads/csw09.pdf
[48] Dyninst Developers. 2016. DynInst - Dynamic Instrumentation Framework. http://www.dyninst.org/parse
[49] Andrew Edwards, Hoi Vo, Amitabh Srivastava, and Amitabh Srivastava. 2001. *Vulcan Binary transformation in a distributed environment.* Technical Report.
[50] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. 2014. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX.
[51] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 901–913.
[52] A. Di Federico and G. Agosta. 2016. A jump-target identification method for multi-architecture static binary translation. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES).* 1–10.
[53] Jack W. Davidson John C. Knight Michele Co Jason D. Hiser Anh Gguyen-Tuong. 2016. *KEVLAR: TRANSITIONING HELIX FROM RESEARCH TO PRACTICE.* Technical Report.
[54] Susan L Graham, Steven Lucco, and Robert Wahbe. 1995. Adaptable Binary Programs. In *USENIX.* 315–325.
[55] Niranjan Hasabnis and R Sekar. 2015. Automatic generation of assembly to IR translators using compilers. In *Workshop on Architectural and Microarchitectural Support for Binary Translation.*
[56] Niranjan Hasabnis and R. Sekar. 2016. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. *SIGOPS Oper. Syst. Rev.* 50, 2 (March 2016), 311–324.
[57] W. H. Hawkins, J. D. Hiser, M. Co, A. Nguyen-Tuong, and J. W. Davidson. 2017. Zipr: Efficient Static Binary Rewriting for Security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* 559–566.
[58] William H. Hawkins, Jason D. Hiser, and Jack W. Davidson. 2016. Dynamic Canary Randomization for Improved Software Security. In *Proceedings of the 11th Annual Cyber and Information Security Research Conference (CISRC '16).* ACM, New York, NY, USA, Article 9, 7 pages.
[59] Kim Hazelwood. 2011. Dynamic binary modification: Tools, techniques, and applications. *Synthesis Lectures on Computer Architecture* 6, 2 (2011), 1–81.
[60] Kim Hazelwood and Artur Klauser. 2006. A dynamic binary instrumentation engine for the ARM architecture. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems.* ACM, 261–270.
[61] Jason Hiser, Anh Nguyen-Tuong, William Hawkins, Matthew McGill, Michele Co, and Jack Davidson. 2017. Zipr++: Exceptional Binary Rewriting. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '17).* ACM, New York, NY, USA, 9–15.
[62] J. K. Hollingsworth, B. P. Miller, and J. Cargille. 1994. Dynamic program instrumentation for scalable performance tools. In *Proceedings of IEEE Scalable High Performance Computing Conference.* 841–850.
[63] Honeywell Inc. 1966. Honeywell Series 200 Operating Systems. Online. http://s3data.computerhistory.org/brochures/honeywell.osorientationmgmt.1966.102646090.pdf
[64] R. Nigel Horspool and Nenad Marovac. 1980. An approach to the problem of detranslation of computer programs. *Comput. J.* 23, 3 (1980), 223–229.
[65] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W Davidson, David Evans, John C Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. 2006. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments.* ACM, 2–12.
[66] Intel Inc. 2016. Intel XED. https://intelxed.github.io/
[67] Josh Poimboeuf Seth Jennings. 2014. Kpatch. http://rhelblog.redhat.com/2014/02/26/kpatch/
[68] Stephen C Johnson. 1990. Postloading for fun and profit. In *Proceedings of the Winter'90 USENIX Conference.* 325–330.
[69] Randy Kath. 1992. The Debugging Application Programming Interface. https://msdn.microsoft.com/en-us/library/ms809754.aspx
[70] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73).* ACM, New York, NY, USA, 194–206.
[71] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2017. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017).* ACM, New York, NY, USA, 412–424.
[72] Johannes Kinder and Helmut Veith. 2008. Jakstab: A static analysis platform for binaries. In *International Conference on Computer Aided Verification.* Springer, 423–427.
[73] Johannes Kinder and Helmut Veith. 2010. Precise static analysis of untrusted driver binaries. In *Formal Methods in Computer-Aided Design (FMCAD), 2010.* IEEE, 43–50.

[74] Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An abstract interpretation-based framework for control flow reconstruction from binaries. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 214–228.
[75] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. 2002. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 191–206.
[76] A. Kiss, J. Jasz, G. Lehotai, and T. Gyimothy. 2003. Interprocedural static slicing of binary executables. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*. 118–127.
[77] B. Korel and J. Laski. 1988. Dynamic Program Slicing. *Inf. Process. Lett.* 29, 3 (Oct. 1988), 155–163.
[78] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, Vol. 13. 18–18.
[79] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. 2014. SoK: Automated Software Diversity. In *2014 IEEE Symposium on Security and Privacy*. 276–291.
[80] James R Larus and Thomas Ball. 1994. Rewriting executable files to measure program behavior. *Software: Practice and Experience* 24, 2 (1994), 197–218.
[81] James R. Larus and Eric Schnarr. 1995. EEL: Machine-independent Executable Editing. *SIGPLAN Not.* 30, 6 (June 1995), 291–300.
[82] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. 2010. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 175–183.
[83] LLVM Compiler Infrastructure. [n. d.]. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html
[84] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200.
[85] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. 2004. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO '04)*. IEEE Computer Society, Washington, DC, USA, 15–.
[86] T. Lundqvist and P. Stenstrom. 1999. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*. 12–21.
[87] Jonas Maebe, Michiel Ronsse, and Koen De Bosschere. 2002. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACTâĂŹ02*.
[88] Amir Majlesi-Kupaei, Danny Kim, Kapil Anand, Khaled ElWazeer, and Rajeev Barua. 2017. RL-Bin, Robust Low-overhead Binary Rewriter. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '17)*. ACM, New York, NY, USA, 17–22.
[89] Cathy May. 1987. *Mimic: a fast system/370 simulator*. Vol. 22. ACM.
[90] Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture. In *USENIX Security Symposium*. https://www.usenix.org/legacy/event/sec06/tech/mccamant/mccamant_html/
[91] Xiaozhu Meng and Barton P. Miller. 2016. Binary Code is Not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*. ACM, New York, NY, USA, 24–35.
[92] Barton P Miller. 2006. Binary Code Patching: An Ancient Art Refined for the 21st Century. NC State University Computer Science Department Seminars 2006-2007. http://moss.csc.ncsu.edu/~mueller/seminar/fall06/miller.html
[93] Robert Muth, Saumya K Debray, Scott Watterson, and Koen De Bosschere. 2001. alto: a link-time optimizer for the Compaq Alpha. *Software: Practice and Experience* 31, 1 (2001), 67–101.
[94] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.
[95] Susanta Nanda and Tzi-cker Chiueh. 2005. A survey on virtualization technologies. *Rpe Report* 142 (2005).
[96] S. Nanda, Wei Li, Lap-Chung Lam, and Tzi cker Chiueh. 2006. BIRD: binary interpretation using runtime disassembly. In *International Symposium on Code Generation and Optimization (CGO'06)*. 12 pp.–.
[97] Nicholas Nethercote. 2004. *Dynamic binary analysis and instrumentation*. Technical Report UCAM-CL-TR-606. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf
[98] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, Vol. 42. ACM, 89–100.
[99] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N Asokan. 2017. CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 259–284.
[100] Trail of Bits. 2017. MCSema. https://github.com/trailofbits/mcsema
[101] Dmytro Oleksiuk. 2014. OpenREIL. https://github.com/Cr4sh/openreil

[102] Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos Keromytis. 2011. Retrofitting security in cots software with binary rewriting. In *IFIP International Information Security Conference*. Springer, 154–172.
[103] Pradeep Padala. 2002. Playing with ptrace, Part I. *Linux Journal* (2002). http://www.linuxjournal.com/article/6100
[104] Mathias Payer, Antonio Barresi, and Thomas R. Gross. 2015. *Fine-Grained Control-Flow Integrity Through Binary Hardening*. Springer International Publishing, Cham, 144–164.
[105] Matt Pietrek. 1994. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. https://msdn.microsoft.com/en-us/library/ms809762.aspx
[106] Manish Prasad and Tzi-cker Chiueh. 2003. A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks. In *USENIX Annual Technical Conference*.
[107] Qemu TCG Developers. 2006. Tiny Code Generator (TCG) Documentation. http://wiki.qemu.org/Documentation/TCG
[108] R. Qiao and R. Sekar. 2017. Function Interface Analysis: A Principled Approach for Function Recognition in COTS Binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 201–212.
[109] Giridhar Ravipati, Andrew R Bernat, Nate Rosenblum, Barton P Miller, and Jeffrey K Hollingsworth. 2007. Toward the deconstruction of Dyninst. *Univ. of Wisconsin, technical report* (2007).
[110] Thomas Reps and Gogul Balakrishnan. 2008. Improved memory-access analysis for x86 executables. In *Compiler Construction*. Springer, 16–35.
[111] Benjamin D. Rodes, Anh Nguyen-Tuong, Jason D. Hiser, John C. Knight, Michele Co, and Jack W. Davidson. 2013. *Defense against Stack-Based Attacks Using Speculative Stack Layout Transformation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 308–313.
[112] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. 1997. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, Vol. 1997. 1–8.
[113] Nathan E Rosenblum, Xiaojin Zhu, Barton P Miller, and Karen Hunt. 2008. Learning to Analyze Binary Computer Code. In *AAAI*. 798–804.
[114] Hex-Rays SA. 2017. IDA-Pro. https://www.hex-rays.com/products/ida/
[115] B. Schwarz, S. Debray, and G. Andrews. 2002. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* 45–54.
[116] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*. Citeseer.
[117] Kevin Scott and Jack Davidson. 2001. Strata: A software dynamic translation infrastructure. In *IEEE Workshop on Binary Translation*.
[118] Kevin Scott, Jack Davidson, and Kevin Skadron. 2001. *Low-overhead software dynamic translation*. Technical Report.
[119] Kevin Scott, Naveen Kumar, Siva Velusamy, Bruce Childers, Jack W Davidson, and Mary Lou Soffa. 2003. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 36–47.
[120] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *USENIX Security Symposium*. 611–626.
[121] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157.
[122] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. 1993. Binary Translation. *Commun. ACM* 36, 2 (Feb. 1993), 69–81.
[123] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *NDSS*.
[124] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2012. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. In *USENIX Annual Technical Conference*. 125–137.
[125] M. Smithson, K. ElWazeer, K. Anand, A. Kotha, and R. Barua. 2013. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *2013 20th Working Conference on Reverse Engineering (WCRE)*. 52–61.
[126] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. *BitBlaze: A New Approach to Computer Security via Binary Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25.
[127] Swaroop Sridhar, Jonathan S Shapiro, Eric Northup, and Prashanth P Bungale. 2006. HDTrans: an open source, low-level dynamic instrumentation system. In *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 175–185.

[128] Amitabh Srivastava and Alan Eustace. 1994. *ATOM: A system for building customized program analysis tools*. Vol. 29. ACM.
[129] K. Richard Pixley Steve Chamberlain, John Gilmore and David Henkel-Wallace. 1992. Binary File Descripor Library 2.29. binutils package. https://sourceware.org/binutils/docs/bfd/
[130] MIPS Computer Systems. 1986. *RISCompiler and C Programmer's Guide*. MIPS Computer Systems, Inc, 930 Arques Ave., Sunnyvale, California 94086.
[131] Ariel Tamches and Barton P Miller. 1999. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Third Symposium on Operating Systems Design and Implementation*.
[132] Elfutils Developer Team. 2017. https://sourceware.org/elfutils/
[133] PaX Team. 2003. PaX address space layout randomization (ASLR). (2003). https://pax.grsecurity.net/docs/aslr.txt
[134] Edward Terry. 2012. Using Liberator. http://ibm-1401.info/1401-Competition.html#UsingLib
[135] TI. 2009. *Common Oject File Format*. Application Report.
[136] David Ung and Cristina Cifuentes. 2001. Optimising hot paths in a dynamic binary translator. *ACM SIGARCH Computer Architecture News* 29, 1 (2001), 55–65.
[137] Cedric Valensi. 2013. *MADRAS: Multi-Architecture BInary Rewriting Tool*. Technical Report.
[138] Valgrind Development Team. 2000. Valgrind. http://valgrind.org/
[139] Ludo Van Put, Bjorn De Sutter, Matias Madou, Bruno De Bus, Dominique Chanet, Kristof Smits, and Koen De Bosschere. 2005. LANCET: A Nifty Code Editing Tool. *SIGSOFT Softw. Eng. Notes* 31, 1 (Sept. 2005), 75–81.
[140] David W Wall. 1992. Systems for late code modification. In *Code GenerationâĂŤConcepts, Tools, Techniques*. Springer, 275–293.
[141] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS'17)*.
[142] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *USENIX Security*. 627–642.
[143] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '17)*. ACM, New York, NY, USA, 319–331.
[144] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS '12)*. ACM, New York, NY, USA, 157–168.
[145] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Securing Untrusted Code via Compiler-agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*. ACM, New York, NY, USA, 299–308.
[146] Richard Wartell, Yan Zhou, Kevin Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. 2011. Differentiating Code from Data in x86 Binaries. In *Machine Learning and Knowledge Discovery in Databases (Lecture Notes in Computer Science)*, Vol. 6913. Springer, 522–536.
[147] Mark N Wegman and F Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 2 (1991), 181–210.
[148] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449.
[149] Mark Weiser. 1982. Programmers Use Slices when Debugging. *Commun. ACM* 25, 7 (July 1982), 446–452.
[150] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-Randomization. In *OSDI*. 367–382.
[151] Liang Xu, Fangqi Sun, and Zhendong Su. 2009. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep* (2009).
[152] Eric Youngdale. 1995. The ELF Object File Format: Introduction. http://www.linuxjournal.com/article/1059
[153] Jonas Zaddach. 2014. Libqemu GIT repository. https://github.com/zaddach/libqemu
[154] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. 2015. VTint: Protecting Virtual Function Tables' Integrity. In *NDSS*.
[155] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Stephen McCamant, and Laszlo Szekeres. 2013. Protecting Function Pointers in Binary. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13)*. ACM, New York, NY, USA, 487–492.
[156] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy*. 559–573.
[157] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. 2014. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*.

ACM, New York, NY, USA, 129–140.
[158] Mingwei Zhang and R Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Usenix Security*, Vol. 13.

Received May 2018; revised November 2018; accepted February 2019