BIRD: Binary Interpretation using Runtime Disassembly

Susanta Nanda Wei Li Lap-Chung Lam Tzi-cker Chiueh
{susanta,weili,lclam,chiueh}@cs.sunysb.edu
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400

Abstract

*The majority of security vulnerabilities published in the literature are due to software bugs. Many researchers have developed program transformation and analysis techniques to automatically detect or eliminate such vulnerabilities. So far, most of them cannot be applied to commercially distributed applications on the Windows/x86 platform, because it is almost impossible to disassemble a binary file with 100% accuracy and coverage on that platform. This paper presents the design, implementation, and evaluation of a binary analysis and instrumentation infrastructure for the Windows/x86 platform called BIRD (*Binary Interpretation using Runtime Disassembly*), which provides two services to developers of security-enhancing program transformation tools: converting binary code into assembly language instructions for further analysis, and inserting instrumentation code at specific places of a given binary without affecting its execution semantics. Instead of requiring a high-fidelity instruction set architectural emulator, BIRD combines static disassembly with an on-demand dynamic disassembly approach to guarantee that each instruction in a binary file is analyzed or transformed before it is executed. It takes 12 student months to develop the first BIRD prototype, which can successfully work for all applications in Microsoft Office suite as well as Internet Explorer and IIS web server, including all DLLs that they use. Moreover, the additional throughput penalty of the BIRD prototype on production server applications such as Apache, IIS, and BIND is uniformly below 4%.*

1. Introduction

A large number of computer system vulnerabilities are due to software bugs. Researchers have proposed various program transformation and analysis techniques to seal security holes. For example, they can prevent buffer overflowing [10, 22], detect tampering of control-sensitive data structures [10, 22], extract sandboxing policy [15], etc. Most of these research projects assume that the source code of the input programs is available. This assumption is unrealistic in practice, because end users rarely have access to the source code of their applications, many of which are created and owned by separate vendors. If these security-enhancing transformations and analysis techniques can be applied to executable binaries directly, it will mark a giant advance in cybersecurity because end users can apply them to their applications on their own. This paper describes the design, implementation, and evaluation of a binary analysis and instrumentation infrastructure for the Windows operating systems running on Intel x86 machines called BIRD (Binary Interpretation using Run-time Disassembly), which can serve as the basis for building these security-enhancing binary transformation tools.

BIRD provides two services to its users with respect to an executable binary: (1) translating the binary file into individual instructions and (2) inserting user-specified instructions into the binary file at specified places. In theory, one can disassemble the input binary into its corresponding assembly language program, insert instrumentation code into it at proper places, and re-assemble it into a new binary. In practice, this is not possible for Windows/x86 binaries because state-of-the-art disassemblers rarely can fully disassemble large Windows/x86 binaries, especially when they do not come with any debug information such as PDB file [1], symbol table, etc. Unfortunately, most commercially distributed Windows/x86 binaries do not carry debug information. Because complete static disassembly is not possible, instrumentation cannot be done completely statically, either.

Some commercial disassemblers such as IDA Pro can report high static disassembly coverage because they are mainly designed to facilitate reverse engineering of executable binaries, and thus can afford to make occasional errors in disassembly results. In contrast, BIRD is designed to support binary instrumentation, and therefore has zero room for disassembly errors. Consequently, BIRD is required to

adopt conservative disassembling techniques that guarantee 100% disassembly accuracy but may achieve lower disassembly coverage than these commercial disassemblers.

To overcome the limitation of static disassembling, BIRD applies disassembly both statically and dynamically. Given an executable binary, BIRD first disassembles it statically to uncover as many instructions as possible, and marks these instructions as *known areas*, leaving the rest as *unknown areas*. At run time, when the program's control is transferred to an unknown area, BIRD disassembles the unknown area as much as it can, and continues with the program execution. By integrating static and dynamic disassembling, BIRD is able to guarantee that every instruction in the input binary be analyzed/transformed before it is executed, while reducing the associated binary interpretation overhead to the minimum.

Instead of rewriting, BIRD takes a *redirecting* approach to binary instrumentation. In theory, this involves nothing but putting the code to be inserted in an unused region, and introducing a jump instruction at the instrumentation point that points to the inserted code. In practice, because a jump instruction takes 5 bytes, it is not always possible to find enough bytes at the instrumentation point for this substitution. BIRD performs both control flow and data flow analysis to guarantee that inserting these jump instructions never affect the program execution semantics. In the worst case, BIRD resorts to the breakpoint instruction (*int 3* on x86) when it cannot find enough bytes to hold the re-directing jump instruction. Furthermore, BIRD performs sophisticated instruction patching and stack fixing to ensure the correctness of the inserted and replaced instructions. BIRD instruments the known areas of a binary file statically, but instruments the unknown area only at run time, i.e., before the program control is actually transferred to them.

The main target of BIRD is commercial Windows/x86 binaries without debug information. It is not meant to be a universal disassembler that can handle binaries which are obfuscated, encrypted/compressed, or in general self-modifying, such as polymorphic virus or tamper-resistant software. Fortunately, most production-mode Windows/x86 application binaries, although not perfectly well-behaved, rarely contain any of these anti-disassembler or anti-debugger code. As a result, the BIRD prototype is able to correctly work on all the applications we have tested, including all applications in the Microsoft Office Suite, Internet Explorer, IIS web server, and system DLLs such as `kernel32.dll`, `user32.dll`, `ntdll.dll`, etc.

2. Related Work

Microsoft's Vulcan [27] applies static disassembling to instrument and optimize binary programs. It extracts an abstract representation from the input binary and the libraries it needs, inserts the instrumentation code to the abstract representation, optimizes it, and converts it into another binary for the target machine. All this happens at the link time. The disassembler in Vulcan depends on the symbol table information, which is available for example in the PDB file generated by Microsoft's Visual C++ compiler. For most commercially distributed Windows applications, including Microsoft Office suite, symbol table information is not available, and therefore Vulcan is not directly applicable. Because the target users of Vulcan are developers, the assumption that symbol table information is available is reasonable. However, BIRD cannot make this assumption because it is meant to be a part of binary transformation tools that are going to be used by end users, who rarely have access to their applications' symbol table information.

There are many link-time optimizers. OM [29] aims at inter-module code optimizations at link time. It relies on the relocation tables available in the object files, and cannot operate on cooked binaries without symbol table information. ATOM [28] is built on OM and further provides a framework for building customized program analysis tools. Plto [23] is closer to BIRD in that it is also targeted at the x86 platform. Plto first collects execution traces from the target binary's runs, and uses them to verify and improve static disassembly results. This approach is used in Strata [26] as well. In contrast, BIRD combines static and dynamic disassembling and completely avoids the trace collection step.

Disassembling Windows/x86 binaries is a difficult problem for two reasons: variable-sized instructions (in contrast to the RISC architecture where instructions are of fixed length) and presence of data inside the code section, some portion of which may not be reachable statically. Earlier disassemblers, like the one illustrated in [24], used a hybrid approach that combines control flow with linear traversal techniques to improve coverage. To further increase coverage, disassemblers apply speculative disassembly techniques [8] that make certain assumptions to continue the disassembling process but later try to confirm them in order to accept the disassembling results. For example, Kruegel et al. [14] apply control flow graph analysis and statistical methods to increase the probability of producing correct disassembled instructions. Similarly, BIRD uses a confidence scoring mechanism (described later in section 3) to measure the validity of the disassembled instructions and then chooses those whose score exceed a certain threshold. While the speculative disassembly approach may be fooled by advanced anti-disassembling techniques [17], our experiences show that it works fine on most commercially distributed Windows binaries. BIRD also leverages from some more sophisticated techniques, such as jump table recovery [7] to further increase the coverage. Ideas similar to function prologs and call targets can also be found in [14].

The ultimate goal of BIRD is to allow a binary transformation tool to apply a proper transformation on every executed instruction before its execution. In this sense, BIRD is related to binary interpreters, which execute a compiled binary on a software simulator or a hardware emulator. Using the idea of software dynamic translation, the Strata project [25] creates an infrastructure on which a virtual execution system can be built. Strata uses a virtual CPU, which is implemented in software, to mimic the target RISC architecture (e.g. fetch/decode/translate) and interpret instructions at run time to discover executed instructions. In addition, the virtual CPU also supports memory, cache, and context management. In contrast, BIRD disassembles unknown instructions at run time without actually ever executing them. Thus, BIRD does not require any high-fidelity instruction set architecture emulator. Valgrind [20], Pin [18] are a few more tools that try to do a similar job; however, these use JIT compilation as their core technology to achieve the translation. DIOTA [19] is another tool that clones the code sections of a binary and applies all the instrumentations on the cloned code pages. By using the original pages for all data accesses, it makes sure any error in instrumentation does not break the program. In contrast, BIRD uses extra memory region for instrumentation code only and does not clone the code pages. The idea of using jump and trap instructions to instrument is already explored in DynInst [5]. BIRD implements it in a different way to improve the performance.

The Embra project [31] develops an emulator of the MIPS R3000/R4000 processors, including the caches and memory systems. To speed up emulation, Embra translates blocks of instructions into native code that simulates the execution of the original block. The system allows the user to dynamically change the level of simulation detail (such as presence of caches, for example), and incurs an execution overhead of 200% to 800%. Also there are several software-based open source emulators currently available, such as Bochs [4] and Plex86 [21].

EEL project [16] develops a system-independent editing model that allows programmers to write binary editing tools in an architecture- and OS-independent manner. It provides a number of abstractions such as an executable, a routine, control-flow graph, etc. EEL relies on the symbol table of a binary to detect the starting addresses of its procedures. However, if the symbol table is not available, EEL employs static disassembly techniques to discover the procedures' entry points. Unfortunately, EEL only runs on SPARC machines under SunOS and Solaris. Etch [11] is a framework for dynamic instrumentation and optimization of Win32/x86 executables. It provides a generalized API that allows custom optimization tools to interact with the core tracing engine. However, the implementation details on Etch are not publicly available.

Dynamo [3] is a binary interpretation and optimization system running on HP PA-8000 machines under HPUX 10.20 operating system. Its key idea is to use a software-based architectural emulator to detect so-called hot traces, i.e. sequences of frequently executed instructions, and optimize them dynamically so that they can run faster. Despite the binary interpretation overhead, Dynamo is able to achieve a non-trivial speedup of 15%-22% for some binaries when compared with their native execution time. Dynamo has been ported to the Win32/x86 platform [6]. It turns out that the Win32/x86 version runs much slower and incurs an overhead of about 30% to 40%. The reasons behind this are lack of documentation on Win32 API and additional implementation complexities that are not present on UNIX platform. Like BIRD, Dynamo can serve as a foundation for security applications. Program shepherding [13] is one such example. By using a disassembler, BIRD reduces the implementation complexity a lot when compared to Dynamo.

3. Disassembly Algorithm

BIRD's disassembler consists of two passes. In the first pass, the disassembler uses *recursive traversal*, which statically traverses the control flow graph of the input binary starting from its main entry point, and discovers all instructions that are reachable through direct branches, i.e., branches whose target address is known statically. All the other bytes in the input binary that are not reachable in the first pass are called *unreachable* bytes. In the second pass, the disassembler assumes some unreachable bytes as instructions, and then performs the same control flow graph traversal from these speculative instructions. In this traversal process, the disassembler accumulates a confidence score on the possibility of an unreachable byte being an instruction. At the end of the second pass, a block of bytes are considered instruction bytes if and only if their score all exceed a certain threshold and its first bytes are indeed the target of some control transfer instruction. BIRD's disassembler does not require any debugger information such as PDB file, and works directly with commercially distributed Windows applications. The only assumptions it makes are

- The byte immediately following a conditional branch instruction starts an instruction.
- No two instructions in the input binary overlap.

However, BIRD's disassembler does not assume the bytes following unconditional jumps, returns or function calls to be instructions. Applying recursive traversal with the above assumptions typically uncover only a small percentage ($<$30%) of the instructions in a PE binary. To improve the coverage without sacrificing accuracy, BIRD's disas-

sembler performs a second-pass recursive traversal assuming the following types of unreachable bytes as instructions:

- Bytes corresponding to an apparent function prologue, i.e., `push ebp, mov ebp, esp`
- Bytes corresponding to the target of a call instruction pattern, i.e., `call x`
- Bytes corresponding to jump table targets.
- Bytes immediately following a jump/call or return.

The second-pass traversal is speculative in nature, and is designed to uncover as many candidate instruction bytes as possible. Not all of these candidate instruction bytes are classified as instruction bytes in the end. Those candidate instruction bytes that lead to instruction overlap or incorrect instruction format are automatically pruned. Then it calculates a confidence score for each second-pass reachable byte using the following heuristic scores: *function prolog* (8), *target of function call* (4), *jump table entry* (2), *target of (un)conditional branch* (1), *bytes after a jump or return* (0), and *data reference* (0). For example, if an instruction byte is part of an apparent function prolog (a jump table entry), its confidence score is increased by 8 (2). When encountering a call instruction in the second pass, the disassembler increases the score of both source and destination bytes of this branch instruction by 4. Because standard compilers typically generate a well-defined prolog for each function, bytes matching a function prolog are considered more likely to be instructions than bytes after a jump or return. Statistically, a call relationship is more reliable than a short branch, because a function call instruction normally takes 5 bytes while a short branch takes only 2 bytes. Although the second-pass traversal uses bytes after a jump or return instruction as starting points, the fact that these bytes are after a jump or return does not contribute to their final score, because it is not uncommon that bytes following a jump or return are actually data.

The confidence score mechanism attempts to capture the essential difference between data and instructions bytes: it is unlikely that data bytes can accumulate multiple evidences that indicate that they are instructions. The final criteria used to determine if a block of bytes correspond to an instruction sequence are the following conditions: (1) their confidence score is above a threshold (currently set to 20), and (2) the first byte of this block correspond to a function prolog, a jump table entry, or a target of a function call. Once BIRD's disassembler decides that a block of bytes correspond to a function, say F, it uses this information to confirm bytes appearing in functions that F calls directly or indirectly as instructions.

To increase the number of possible starting points for the speculative recursive traversal in the second pass, BIRD's disassembler performs additional analysis to recognize constructs such as a jump table, which is a data block composed of a sequence of target addresses, and is used to support control constructs such as `switch` statements in C. Normally, a program using a jump table calculates an index of the jump table, and then takes an indirect jump whose target address is the jump table entry corresponding to the index. To recognize jump tables, BIRD's disassembler starts with memory references of the form of a base address plus four times a local variable, and then examines the region surrounding the base address to identify a continuous sequence of words each of which is both aligned and pointing to a valid instruction. Because an instruction immediately preceding a jump table could also include one or two addresses as its operands, entries in the discovered sequence, except the first two, are marked as jump table entries. The nature of the first two entries will be determined when the nature of their preceding bytes is determined later on.

In addition, BIRD's disassembler leverages the binary format to discover more starting points. First, some data embedded in the code section could be identified from the binary format. For example, the location of a Windows binary's import address table is specified in the binary's header. Second, a binary's export table entries, which are locations of exported functions or variables, indicate whether the corresponding bytes are instructions or data. Third, the relocation table, which typically comes with DLLs, greatly simplifies the task of identifying jump tables, as each jump table entry should have a corresponding relocation entry. Relocation table entries could also be used to check validity of candidate instructions. For example, a relocation table entry should never point to an instruction without data/address reference.

4. BIRD Run-Time Architecture

4.1. Overview

Figure 1 shows how BIRD interprets each instruction in a Windows binary before it gets executed. Given an input binary, BIRD first disassembles it statically as much as it can, and label those parts that are successfully disassembled as *known areas* (KA), and those that are not as *unknown areas* (UA). Any application-specific instrumentation is statically applied to the KAs only. BIRD disassembles the unknown areas at run time by intercepting control transfers from known areas to unknown areas. The only instructions in the known areas that could jump to unknown areas are indirect branches, which are control transfer instructions whose target is computed using contents of a memory location and/or registers, e.g., indirect jump and call, and return instruction. In addition, control could be transferred to an unknown area because it contains callback functions and exception handlers, which are invoked by the kernel. We will focus on indirect branches first.

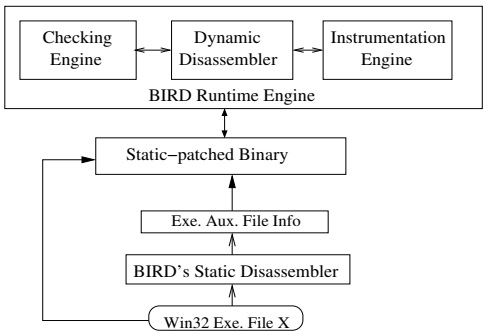

Figure 1: **BIRD's architecture consists of a static disassembler and a run-time engine, which in turn consists of a dynamic instrumentation and a dynamic disassembler. During an application's startup, BIRD's run-time engine is loaded into the application's address space as a DLL. BIRD patches all indirect branches so that it can intercept them at run time and dynamically disassemble the statically unknown areas.**

BIRD takes control at indirect branches by replacing them with a jump to a special function `check()`, which is the core of BIRD's run-time engine and performs the following functions:

- Calculate the target address of the replaced indirect branch,
- Check if the target falls into an UA, and if so invoke the dynamic disassembler to convert the UA or part of it into a KA,
- Perform application-specific instrumentation on the newly discovered instructions, and
- Execute the replaced instructions.

The output of BIRD's static disassembler consists of (1) a list of UAs (UAL), and (2) the locations of the indirect jumps/calls and how they should be patched (IBT). Both UAL and IBT are appended to the input binary as a new data section [9], and read in at startup time and stored in main memory as a hash table. Indirect branch target is computed by executing a push instruction with the data operand same as that of the original instruction (for example, from `call [eax+4]` to `push [eax+4]`) and then reading from the stack. To determine if the computed target address falls into an UA, `check()` consults the UAL through a hash look-up. To speed up the common case in which the target falls into a KA, `check()` also maintains a KA cache, which is also organized as a hash table. After invoking the dynamic disassembler on an UA, the UA could totally vanish if all of its bytes are explored, could become smaller if its tail is explored, or could be broken into two disjoint pieces. `Check()` updates the UAL accordingly.

To ensure that the instructions replaced by a call to `check()` are executed inside `check()` in exactly the same architectural context as the original case, `check()` saves the original stack and register state once it takes control and restores them before executing the replaced instructions. After these replaced instructions are done, the control is transferred back to the corresponding instrumentation point.

The initialization routine and `check()` of BIRD's run-time engine is organized as a DLL called `dyncheck.dll`, and is completely independent of the applications being instrumented. By modifying the import table of the instrumented application, `dyncheck.dll` is automatically loaded when the application starts up. Because the initialization routine of a DLL always gets control when the DLL is loaded [1], this enables BIRD to read in the application's UAL and IBT and initialize required data structures before the program's main function starts. As a DLL function, `check()` can directly access an application's UAL and IBT using symbolic names. Since many real-world Windows applications use DLLs extensively, BIRD needs to support arbitrary DLLs. More specifically, it requires all such DLLs to be disassembled a priori so that their UAL and IBT are available beforehand, and modifies their initialization routine to read in their own UAL and IBT in exactly the same way as executable files. Because a program's import table may be immediately followed by some other data, it is not always possible to increase its size directly. To solve this problem, BIRD keeps the old import table, creates a new import table that contains the original import table entries and any new entries we want to add, and modify the import table address field in the binary's header to point to the new import table.

4.2. Callback Functions

Windows provides several mechanisms by which a user-level application can supply the entry point of a callback function that the kernel can call upon certain event(s). For these callback functions, there are no explicit call sites inside the applications. Windows supports three types of callbacks: *exception handler*, *callback function*, and *asynchronous procedure call* (APC). They all work in a similar way, so we will focus only on callback functions. When the kernel invokes a callback function, it switches context and jumps to `KiUserCallbackDispatcher()` in the `ntdll.dll` library. `KiUserCallbackDispatcher()` then calls a function in `user32.dll` to look for the corresponding user-supplied function in a special data structure, which was initially populated when the application registered the callback, and to invoke the callback function if found. When the callback function is finished, the `user32.dll` routine traps back to the kernel for further processing by executing the instruction `int 0x2B`.

Because a callback function is invoked from a

user32.dll routine through a function pointer, BIRD can analyze/transform the instructions in callback functions before they are executed without any additional mechanisms. However, BIRD cannot intercept the control transfer from the kernel to KiUserCallbackDispatcher() when it invokes a callback function. Fortunately, this interception is not necessary because BIRD can statically disassemble functions in system DLLs (ntdll.dll, kernel32.dll, and user32.dll) with the help of their export tables, which contain symbol and location information for every exported DLL function. Since each of these routines that the kernel jumps to are exported by ntdll.dll, BIRD can statically analyze/transform the instructions and does not need to intercept the kernel-to-user control transfer at run time. Although export tables could potentially contain entries corresponding to variables as well, ntdll.dll does not contain any such cases. Now that BIRD has its control on ntdll.dll functions, it disassembles and transforms/patches user32.dll as it does with other DLLs and handles the indirect calls to user-supplied callback functions in exactly the same way as normal indirect calls.

Exception handlers can potentially change the control flow as a side effect of handling the exception. They typically use the EIP register to indicate where in the application should the kernel return control to. Consequently, when BIRD intercepts the return instruction of an exception handler, it uses the EIP register rather than the return address as the target of the return instruction and invoke the dynamic disassembler if the target happens to fall in an UA.

4.3. Speculative Dynamic Disassembly

When check() encounters an UA through an indirect branch, it invokes the disassembler to uncover as many instructions as possible from that UA. More specifically, the disassembler scans through the UA starting from the indirect branch's target address, and keeps on disassembling instructions until it reaches a control transfer instruction that jumps to some KA. Any code area that is uncovered in this process is merged into existing KAs and the UAL is updated. In addition, all the indirect branches in the new area are replaced either by a call to check(), or a breakpoint (int 3). This allows BIRD to intercept at all these newly discovered indirect branch instructions.

To reduce performance penalty, dynamic disassembler is simplified in two aspects: (1) there is no second pass, and (2) all short indirect branches are replaced by breakpoints. The first results in more number of calls to disassembler at the runtime, while the later causes more context switches due to breakpoints. Our experiences suggest that when the coverage of BIRD's static disassembler on an application is low, the application's execution time increases dramatically because of the additional int 3 instructions.

In general, to achieve high disassembly accuracy, it requires more conservative disassembling strategies, which imply low disassembly coverage and thus higher run-time overhead. To attain high disassembly accuracy while minimizing the run-time overhead, BIRD uses a speculative disassembly technique. When BIRD's static disassembler disassembles an input binary, it is conservative when outputting the unknown area list (UAL) to increase the disassembly accuracy. However, it keeps the disassembled results of the unknown areas, even if it is not sure whether they are correct. At run time, when BIRD's dynamic disassembler is invoked due to an indirect branch instruction that jumps to an UA, it first checks whether the UA's speculatively disassembled result also thinks the branch's target address starts an instruction. If so, the dynamic disassembler simply borrows the corresponding portion of the UA's speculatively disassembled result without performing any disassembling; otherwise it disassembles the UA on its own. Consequently, BIRD can leverage the statically disassembled results that cannot be proven to be 100% accurate by confirming at run time that their underlying assumptions are correct. Because BIRD produces these disassembled results statically, it can afford to use a more sophisticated instrumentation scheme, as described in the next subsection, to replace indirect branch instructions with call instructions to check(), and greatly reduce the number of int 3 instructions executed and thus the overall run-time overhead.

4.4. Binary Instrumentation

BIRD also provides a binary instrumentation service for application developers to modify existing binaries. In fact, BIRD itself also needs to modify the input binaries for intercepting indirect calls/jumps. Because BIRD does not necessarily have access to the entire assembly representation for an input binary, it cannot instrument the input program at the assembly level and re-assemble the resulting program. Instead, it instruments an input binary directly at the binary level by replacing the instruction at the instrumentation point with a branch instruction to the user-supplied instrumentation code, which is followed by the replaced instruction(s) and in the end transfers the control back to the instrumentation point.

Although conceptually simple, BIRD's instrumentation algorithm is surprisingly difficult to implement in practice for the following two reasons. First and foremost, if the instruction to be replaced does not have enough space to accommodate a branch instruction (typically 5 bytes long) to the instrumentation code, it is not always possible to find enough bytes surrounding it. In the example in Figure refreplacement:fig, the *jmp* instruction at 401308 is 6 bytes long which is long enough to be replaced by the call in-

struction to `check()`. What if the instrumentation point corresponds to a short instruction, e.g., the instruction at address 4012ef? Such cases are not rare. As an example, when BIRD intercepts indirect calls/jumps, in many cases it needs to replace a 2-byte-long (short) indirect branch like `call eax` by a 5-byte call instruction to `check()`. Our measurements show that the fraction of short indirect branches among all indirect branches is between 30% to 50% in both static and dynamic counts. Second, if execution of the replaced instruction(s) is required, it is essential to preserve the same execution context for these instructions when the control is in the instrumented code. In the example, the instruction at 49a03d (`add edx, edi`) would depend on the registers `edx` and `edi`. Therefore, to ensure correctness, register values should be preserved for replaced instructions.

When the instruction at the instrumentation point is shorter than 5 bytes, additional bytes could come from the first one or two instructions immediately following the instruction at the instrumentation point as long as doing so does not affect the program's execution semantics. In general, an instruction is *safe* to be replaced if it is not the target of any branch instruction. BIRD takes this one step further: *it is safe to replace an instruction as long as it is not the target of any direct branch in the same application.* Although it might appear to be unsafe to replace an instruction if it is the target of an indirect branch, it is safe to do so in BIRD because BIRD intercepts every indirect branch. At run time, when BIRD finds out that the target of an indirect branch goes to a replaced instruction, it executes these replaced instruction(s) until the control jumps out of the replaced bytes. The example in Figure 2 illustrates this point, where the function contains two indirect branches, at addresses 0x4012ef and 0x401308. Because the first one (0x4012ef) is a short branch, BIRD merges the following two instructions to create space for a call instruction to `check()`. The second one (at 0x401308) is 6-byte long and has enough space, but can potentially jump to any instructions between 4012ef and 4012f3. However, as BIRD intercepts this indirect jump, BIRD can check if the target address lies within [4012ef, 4012f3] and if so, could directly execute the original instructions in the target address before jumping to the following instruction at 0x4012f5. For instance, if the target of the indirect jump at 0x401308 is 0x4012f3, the sequence of action that BIRD takes is: (1) execute `jmp [ebx]4`, (2) copy the original two bytes from 0x4012f3 to some address, (3) execute `mov eax, edx`, and (4)jump to 0x4012f5. The above algorithm works quite effectively in practice, as most short instructions at the instrumentation points can indeed be safely merged with the following few instructions to create enough space for a 5-byte call instruction to `check()`.

When BIRD's static disassembler cannot find any

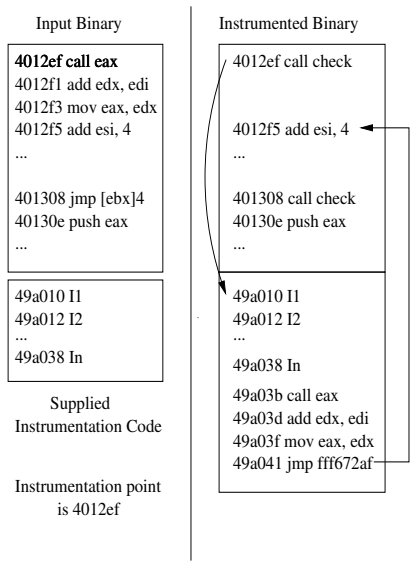

Figure 2: **This example illustrates how BIRD can replace instructions that are potential targets of an indirect branch (in this case `jmp [ebx]4`) because its run-time engine intercepts every indirect branch and thus can always run the original target instructions even when they are replaced.**

safe bytes to insert the 5-byte branch instruction, it replaces the short instruction at the instrumentation point with a 1-byte breakpoint instruction (`int 3`, opcode 0xcc), whose exception handler in turn calls `check()`. In Windows, a program could register multiple handlers associated with an exception, which are invoked in the order in which they are registered. To ensure BIRD's `int 3` exception handler is the first to handle all `int 3` instructions BIRD puts in, BIRD intercepts the `KiUserExceptionDispatcher()` function in `ntdll.dll` and always invokes BIRD's breakpoint handler for BIRD's `int 3` instructions.

The detailed flow of how BIRD replaces an indirect branch instruction with a branch instruction to its run-time engine is illustrated in Figure 3(A). At the instrumentation point, a jump instruction takes the control to a stub, which consists of an instruction that computes the target address of the indirect branch instruction, a call instruction to a `check()`, the original indirect branch instruction, possibly a sequence of replaced instructions, and finally a jump instruction back to the instrumentation point. The `check()` routine itself consists of two components, one for register state saving and restore and the other (called `real_chk()`) for determining if the instrumented indirect branch jumps to a known or unknown area. Before calling `real_chk()`, `check()` looks up its known area cache first. This stub is statically generated and allocated for each instrumented indirect branch instruction. Because replaced instructions are

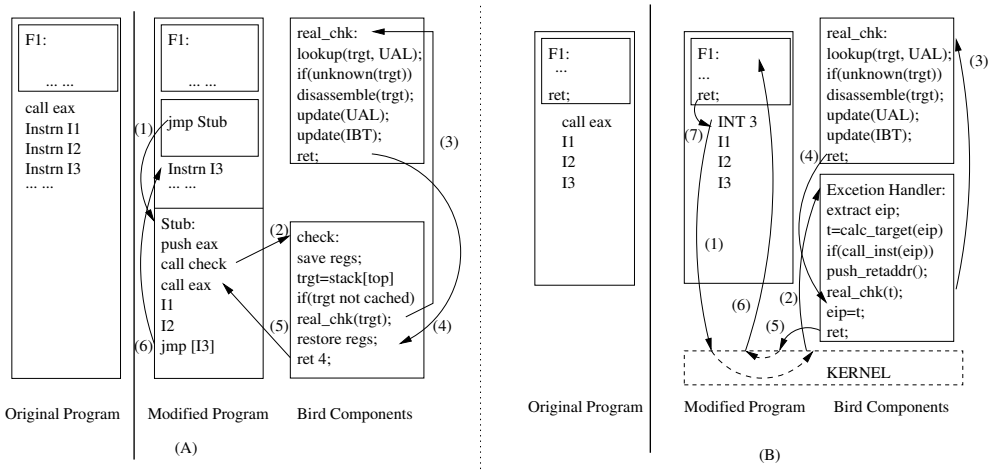

Figure 3: **In (A), BIRD instruments an indirect branch instruction, in this case `call eax`, by replacing it and its following two instructions with a jump instruction to a stub, which calls `check()` and optionally executes the replaced instructions. The `check()` routine in turn calls `real_chk()` to determine if the target falls into an unknown area and if so invoke the dynamic disassembler. In (B), BIRD instruments the indirect branch instruction `call eax` by replacing it with an `int 3` instruction. BIRD provides an exception handler for the breakpoint exception, and performs similar functions as the stub and `check()` combined. The numbers on the arcs show the order in which the control is transferred among various components. In this case, the target function of `call eax` is `F1()`.**

moved from their original location, BIRD needs to update relocation information if they happen to be targets of relocation, and convert them into position-independent code, i.e., turning all relative offsets into absolute addresses. However, some instructions can only take relative addresses but not absolute addresses. and therefore need to be transformed into two instructions. For example, a relative-offset instruction `jecxz 100` at address 1000 has an absolute target address of 1102 (offsets are always added to the address of the following instruction), and needs to be converted to something like `jecxz 10; ..., jmp 1102`, where the `jmp` instruction is 10 bytes away from the `jecxz` instruction and comes after the final jump in the stub. Putting the absolute jump at the end makes sure the execution path remains correct if the branch is not taken, i.e. ECX is non-zero.

If an instrumented indirect branch instruction is replaced with an `int 3` instruction, BIRD's `int 3` exception handler will perform similar functions as a stub and `check()` combined, as shown in Figure 3(B). The only difference is that to "execute" the instrumented indirect branch, the exception handler sets the `EIP` register to the branch's target before it returns to the kernel, and pushes a proper return address to the stack if the indirect branch is an indirect call. Since dynamically discovered indirect branches are always replaced with `int 3` instructions, they do not require any stubs and thus no stubs are generated dynamically.

Application	Code Size(KB)	Disassembled(KB)	Coverage	Accuracy
lame-3.96.1	241.6	233.6	96.70%	100%
ncftp-3.1.8	192.5	162.4	84.39%	100%
putty-0.56	369.1	354.8	96.12%	100%
analog-6.0	311.2	276.1	88.71%	100%
xpdf-3.00	319.4	275.1	86.12%	100%
make-3.75	122.8	117.3	95.50%	100%
speakfreely-7.2	229.3	160.2	69.97%	100%
tightVNC-1.2.9	180.2	135.0	74.90%	100%

Table 1: **Disassembly coverage and accuracy for applications with source code: The disassembly accuracy is computed based on comparison between the output from BIRD's disassembler and the assembly code generated by Visual C++ 6.0.**

4.5. Extensions

BIRD's instrumentation architecture can also be extended to support arbitrary self-modifying code. There are three modifications to the original BIRD architecture. First, BIRD needs to intercept direct branches as well as indirect branches. This ensures that all branch targets are properly instrumented before their execution. Since direct branch targets are no more constsnt, static disassembly can only work for the first block of the binary, so disassembling is done mostly dynamically. Second, the dynamic disassembler needs to be more aggressive in identifying replaced instructions during binary instrumentation, in order to reduce the number of `int 3` instructions and thus the associated performance penalty. Third, when the target of a direct or indirect instruction falls into a read/write page, BIRD needs

to invoke the dynamic disassembler on the target block even if the target block has been disassembled previously. More specifically, every time BIRD's run-time engine disassembles a block of bytes, it marks the page containing the block as read-only. If the application tries to modify the page, it generates a protection fault, which BIRD's run-time engine intercepts and turns that page into read-write. The current BIRD prototype only implements a subset of the above architecture and can successfully run Windows applications that are transformed by binary compression tools such as UPX [30].

5. Performance Evaluation

5.1. Disassembly Accuracy and Coverage

To evaluate the effectiveness of the disassembly algorithms used in BIRD's static disassembler, we use two sets of programs, one with source code and the other without. The first set of programs, shown in Table 1 come with source code and hence are readily compilable under Visual C++ 6.0. They are compiled with the options to generate the intermediate assembly representation and the program database (PDB) file with detailed symbol information. These options do not affect the final binary file outputs. Then we apply BIRD's disassembler to each application's binary file *without using any PDB information* to produce an assembly output, which is then compared with Visual C++ compiler's assembly output. Because BIRD is designed to support binary instrumentation, its disassembly output has to be absolutely accurate, i.e., bytes in the binary file that are identified as instructions must be indeed instructions. Disassembly accuracy is defined as the fraction of instructions from BIRD's disassembler output that match the ground truth, in this case, the output of the Visual C++ compiler's assembly code. To measure disassembly accuracy, we first extract function names from the generated PDB file and identify each function's instructions in Visual C++ compiler's assembly file. Because the PDB file also contains each function's starting address, we then use this information to locate each instruction's absolute address in the binary file. This result serves as the ground truth against which the output of BIRD's disassembler is compared. However, there could be instructions that BIRD's disassembler produces from the binary file that are not present in Visual C++ compiler's assembly output. For instance, statically linked libraries that go into the binary do not have their source code available (an example is `libc.lib`, a Microsoft visual C runtime library). Such instructions, as well as other no-op instructions (e.g. `mov eax eax`), are just ignored when comparing these two assembly outputs.

Table 1 shows the disassembly accuracy and coverage for several Windows applications compiled with Visual C++. Disassembly coverage is defined as the percentage of bytes in the input binary file that the disassembler has successfully identified as instructions or data. For all programs tested, the accuracy of BIRD's disassembler is 100%. This demonstrates that the heuristic scoring mechanism used in BIRD's disassembler is not overly aggressive. As expected, the disassembly coverage is not 100%, and ranges from 69% to 96%. This demonstrates the need for BIRD's dynamic disassembling approach.

We then apply BIRD's disassembler against several popular Windows applications, whose source code is not available, and measure their coverage. The results are shown in Table 2. Because the ground truth is unavailable, we verify the accuracy of BIRD's disassembler in two ways. First, we run these applications under BIRD and monitor their execution behavior for any major errors. Because BIRD instruments an application based on the disassembly result, disassembly errors lead to incorrect instrumentation, which in turn may result in execution errors or crash. Second, we compare BIRD disassembler's output with the output of IDApro, which is arguably the most popular commercial disassembler, and check if bytes that BIRD's disassembler identifies as instructions are also instructions as far as IDApro is concerned. It is possible IDApro identifies more instruction bytes than BIRD because it does not require 100% disassembly accuracy. Throughout these tests, we are yet to find any disassembly errors in each application in Table 2. Because all of these applications have a user interface component, which embeds a substantial amount of data in the code section, it is more difficult to disambiguate them statically. As a result, the corresponding disassembly coverage, which ranges from 53% to 78%, is lower than the batch programs in the first set.

To evaluate the effectiveness of different disassembling heuristics, we measure the incremental improvement in disassembly coverage from each heuristic. The results are shown in Table 2. Since pure recursive traversal without any assumptions usually achieves very low coverage (less than 1%), we show the result of extended recursive traversal, which speculatively disassembles bytes from instructions following call instructions. Even then, it can only achieve a coverage result between 6% to 36%. Exploiting function prolog pattern significantly boosts the disassembly coverage because existing compilers generate well-defined function prologs, and accordingly BIRD's disassembler assigns a higher confidence score. Recognizing bytes corresponding to function call targets and jump table entries further improves the disassembly coverage. However, assuming bytes immediately following a jump or return start an instruction doesn't seem to do any good here, because compilers indeed put data right after a jump and return instruction. This is why BIRD's disassembler gives a zero score to this heuristic. We use this heuristic only as a way to dis-

Application	Code Size (bytes)	Extended Recursive Traversal	Function Prologue Pattern	Func. Call Target	Jump Table Entry	Spec. Jump & Return	Data Ident.	Original Startup Delay	BIRD Startup Penalty
MS Messenger	1052672	13.36%	58.04%	59.81%	66.02%	66.38%	**74.62%**	857M	11.25%
Powerpoint	4136960	6.65%	34.84%	40.34%	46.51%	47.25%	**53.58%**	2568M	32.23%
MS Access	4145152	27.19%	56.31%	58.80%	62.24%	62.62%	**65.29%**	3186M	22.56%
MS Word	7864320	36.35%	71.19%	71.38%	76.35%	76.84%	**78.06%**	1887M	12.56%
Movie Maker	638976	5.11%	63.59%	68.94%	72.69%	73.88%	**74.30%**	1892M	14.67%

Table 2: **The incremental contributions of different heuristics to the overall disassembly coverage for Windows binaries. The Speculative Jump/Return technique applies linear sweeping to bytes immediately following a jump/return instruction. The numbers in bold are the final coverage percentages. The Original Startup Delay is in terms of CPU cycle (M means 10^6) whereas the BIRD Startup Cost is in terms of additional percentage overhead.**

Appl.	Orig. Ex(G)	BIRD Ex(G)	Init Ovhd	DDO	Chk Ovhd	Total Ovhd
comp	0.19	0.24	14.9%	0.1%	0.2%	15.2%
compact	10.28	10.94	6.4%	0.0%	0.0%	6.4%
find	7.44	7.91	5.6%	0.0%	0.6%	6.2%
lame	1.19	1.34	12.0%	0.0%	0.0%	12.0%
sort	0.26	0.31	16.1%	0.4%	1.4%	17.9%
ncftpget	1.06	1.09	3.4%	0.0%	0.0%	3.4%

Table 3: **Increase in execution time for six batch programs under BIRD are due to initialization of DLLs and data structures (Init Ovhd), checking at all the indirect branches (Chk Ovhd), and invoking dynamic disassembler on unknown areas (D.D.O, i.e. Dynamic Disassembly Overhead). Breakpoint handling overhead is close to 0 in these cases and are not shown here. Original and BIRD execution times are both expressed in number of CPU cycles (G means 10^9).**

cover more code, but it turns out that most of these bytes were uncovered by earlier heuristics already. The ability to identify data has a noticeable effect on disassembly coverage as it enables early pruning of non-code bytes. As a result, it achieves non-negligible improvement in disassembly coverage for some applications, e.g., more than 8% for MS Messenger.

5.2. Run-Time Overhead

The current BIRD prototype can successfully work on large Windows applications, including MS Office applications, Internet Explorer, Acrobat reader, etc. However, to characterize BIRD's run-time overhead for these interactive applications, we measure the overhead incurred during their startup, i.e., the time between when a program is started and the time when it is ready to receive inputs from the user, on a Pentium-IV 2.8GHz/256MB Windows XP machine. We start a timer just prior to `CreateProcess()` and stop it just after `WaitForInputIdle()`, which corresponds to the time when the application is ready to receive user inputs. The last two columns of Table 2 show that the startup delay of these interactive applications is increased by 10% to 35%. Although the startup delay penalty appears substantial, the bulk of this penalty only occurs at program initialization time but not at run time, because it is related to DLL loading and relocation. That's why our own usage experiences show that the interactivity of these applications is not affected at all when they run under BIRD.

Next, we run a set of six batch programs on a Pentium-IV 2.8GHz/256MB Windows XP machine under BIRD and measure the increase in program execution time. These programs are *comp* (comparing two 4.4MB files), *compact* (compressing a set of twelve binary times in a directory), *find* (finding a given string from a 500KB DLL file), *lame* (converting a wav audio file to mp3 format), *sort* (sorting a 500KB ascii file), and *ncftp* (getting a 1KB file through file transfer protocol from a remote machine). The performance overhead for these applications come from several sources. The initialization overhead includes the time spent on reading/initializing UAL and IBT from disassembler output files and the relocation overhead for system DLLs because they are modified. The checking overhead, shown in the *Check Overhead* column of Table 3, represents the overhead of invoking `check()`. The runtime disassembly overhead, shown in the *Dyn. Disasm. Overhead* column, shows the overhead of invoking the dynamic disassembler on the statically unknown areas. Breakpoint handling overhead is not shown here because of extremely small penalty (less than 0.005%) involved in these batch programs.

Table 3 shows the break-down of the performance overhead of these six batch programs running under BIRD. The initialization overhead dominates all other types of overheads, because the loader needs to load the additional DLL, `dyncheck.dll`, which implements BIRD's run-time engine, and relocate system DLLs. Because BIRD instruments a DLL in the same way as it instruments an executable file, the instrumentation could increase a DLL's size. The Windows OS tends to load system DLLs in their preferred locations. When some DLLs grow in size and cannot fit into the originally allocated space, the loader has to relocate them. This initialization overhead has no impact on an application's run-time performance after the initialization stage and translates to high performance penalty percentage only for short-running applications, which do not

Application	Dynamic Disassembly Overhead	Dynamic Check Overhead	Breakpoint Handling Overhead	Total Ovhd
Apache	0.12%	0.73%	0.07%	0.9%
BIND	0.26%	2.33%	0.51%	3.1%
IIS W3 service	0.15%	0.83%	0.12%	1.1%
MTSPop3	0.09%	1.31%	0.00%	1.4%
Cerberus FTPD	0.12%	0.94%	0.14%	1.2%
BFTelnetd	0.39%	0.67%	0.44%	1.5%

Table 4: **Detailed measurements of binary instrumentation effects on commercial server applications. The Dynamic Disassembly Overhead refers to the performance overhead due to invocation of dynamic disassembler. The Dynamic Check Overhead refers to the performance overhead due to call to `check()`. The Breakpoint Handling Overhead refers to the performance overhead due to `int 3` instructions that BIRD inserts.**

use the loaded DLLs for a sufficiently long period to amortize the incurred cost. Despite the significant initialization overhead, BIRD still performs much better when compared with exception-based binary interpretation approaches such as Valgrind [20].

Finally, we measure the throughput penalty of several production-mode network server applications when running under BIRD. More specifically, we send a fixed number of requests (2000 in these results reported below) to each server application, and measure the throughput difference between when it runs under BIRD and when it runs natively. Each request fetches a 1KByte HTML file, FTP file, mail message, or DNS records, depending on the server being tested. The server application runs on a Pentium-IV 2.8GHz/256MB Windows XP machine, whereas the client is a Celeron 500MHz/192MB RedHat 7.2 machine. The results in Table 4 show that the throughput penalty of BIRD is below 4%. The *initialization overhead* is ignored as it does not affect the throughput penalty measurement. In general, the performance overhead of BIRD does not come from dynamic disassembler invocation or breakpoint handling. It is the number of dynamic checks and lookups (when there is a known area cache miss) involved that matters the most. As an application uses more DLLs, it increases the number of checks and slows down each check. In the case of BIND, it incurs a significant amount of check overhead because a larger number of checks at run time and a higher per-check lookup overhead due to cache misses. In contrast, even though the number of dynamic checks for IIS is comparable to BIND, its per-check lookup overhead is lower and as a result its total performance overhead is also smaller.

6. An Application: Foreign Code Detection

To demonstrate the effectiveness of BIRD we apply it to build a *foreign code detection* system, which aims to detect un-authorized control transfers to injected or existing code in an application run. There are several techniques by which an attacker can inject a piece of code into a running process, and steer the process' control to the injected code. Buffer overflow attacks and format string attacks are two such techniques. One technique to stop these code-injecting attacks is a program execution mechanism that can distinguish between instructions in an application's binary file and instructions that are injected at run time. The foreign code detection system (FCD) distinguishes between native and injected instructions based on their *location*, rather than *content* as employed by [12]. Because FCD assumes its target applications do not contain any self-modifying code, it can statically identify all the code sections, including DLLs, and safely mark them as read-only. At run time, when a control transfer instruction attempts to jump to an area outside the code sections, the target must be an injected instruction. FCD leverages BIRD's interception mechanism to perform the check that the target address of each indirect branches is always within the code sections. In addition, by moving the entry points of sensitive DLL functions, FCD can also detect return-to-libc attacks [2].

7. Conclusion

Binary analysis and instrumentation is a key enabling technology for securing application binaries through program transformation. However, perfect static disassembly is almost impossible for commercially distributed binaries on the Windows/x86 platform, because they do not come with any debugger information such as symbol table, relocation table, etc. Microsoft's Vulcan requires a binary's full PDB file in order to completely disassemble it. As a result, existing security-enhancing program transformation techniques rarely can be applied to commercial Windows applications. This paper describes the design, implementation and evaluation of a binary analysis and instrumentation infrastructure called BIRD (Binary Interpretation using Run-time Disassembly), which combines static and dynamic disassembly in a novel way to achieve both 100% coverage/accuracy and low run-time overhead for Windows/x86 binaries. As a result, we expect BIRD to become a key building block in future software security systems.

The current BIRD prototype can successfully run large Windows applications such as Microsoft Office suite, Internet Explorer, IIS, Acrobat Reader, etc., and the additional non-startup runtime performance overhead is under 5%. By leveraging disassembling techniques extensively, BIRD is much simpler in design/implementation complexity when compared with other similar systems that require a high-fidelity instruction set architecture emulator. For example, the current BIRD prototype takes fewer than 12 graduate student months to complete. To demonstrate the usefulness of BIRD, we successfully develop a foreign code detection

system based on BIRD that guarantees no foreign code injected at run time can be executed in the protected application, and that no un-authorized control transfers to sensitive DLL functions are possible. This demonstration application itself takes fewer than 5 months to complete. Finally, as part of this project's development efforts, we performed a comprehensive study on the effectiveness of various disassembling techniques in terms of their coverage and accuracy. We believe this is the first time such accuracy and coverage results ever appear in the open literature.

We are currently enhancing the instrumentation API for BIRD so that it can be used as a general binary instrumentation system. We are also applying BIRD to other security applications such as system call pattern extraction, attack signature extraction, and automatic post-intrusion repair. Finally, we are extending BIRD according to the architecture described in Section 4.5 so that it can successfully instrument general self-modifying binaries with low overhead.

References

[1] Microsoft msdn library. http://msdn.microsoft.com/library/.

[2] phrack. http://www.phrack.org/.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[4] Bochs. Bochs: The cross platform ia-32 emulator. http://bochs.sourceforge.net/, 2001.

[5] B.R.Buck and J.K.Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[6] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.

[7] C. Cifuentes and M. V. Emmerik. Recovery of jump table case statements from binary code. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, page 192, Washington, DC, USA, 1999. IEEE Computer Society.

[8] C. Cifuentes, M. V. Emmerik, D. S. D Ung, and T. Waddington. Preliminary experiences with the use of the uqbt binary translation framework. In *Proceedings of the Workshop on Binary Translation*, 10 1999.

[9] P. Dabak, M. Borate, and S. Phadke. Undocumented Windows NT. M and T Books, October 1999.

[10] C. C. et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *7th USENIX Security Symposium*, 1998.

[11] T. R. et al. Instrumentation and optimization of win32/intel executables using etch, 1997.

[12] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280. ACM Press, 2003.

[13] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, 2002.

[14] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium 2004*, pages 255–270.

[15] L. Lam and T. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In *Seventh International Symposium on Recent Advances in Intrusion Detection*, September 2004.

[16] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1995.

[17] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM Press, 2003.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[19] J. Maebe, M. Ronsse, and K. D. Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Proceedings of the 4th Workshop on Binary Translation*, 2002.

[20] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.

[21] Plex86. Plex86 x86 virtual machine. http://savannah.nongnu.org/projects/plex86.

[22] M. Prasad and T. cker Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX Annual Technical Conference*, pages 211–224, 2003.

[23] B. Schwarz, S. Debray, and G. Andrews. Plto: A link-time optimizer for the intel ia-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, Sept 2001.

[24] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 45. IEEE Computer Society, 2002.

[25] K. Scott and J. Davidson. Strata: A software dynamic translation infrastructure. In *Proceedings of the 2001 Workshop on Binary Translation*, 2001.

[26] K. Scott, J. Davidson, and K. Skadron. Low-overhead software dynamic translation. Technical Report CS-2001-18, July 2001.

[27] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary Transformation in a Distributed Environment. Technical Report MSR-TR-2001-50, 2001.

[28] A. Srivastava and A. Eustace. Atom: a system for building customized program analysis tools. *SIGPLAN Not.*, 39(4):528–539, 2004.

[29] A. Srivastava and D. W. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, December 1992.

[30] UPX. the ultimate packer for executables. http://upx.sourceforge.net/.

[31] E. Witchel and M. Rosenblum. Embra: Fast and flexible machine simulation. In *Measurement and Modeling of Computer Systems*, pages 68–79, 1996.