FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCE

AUTOMATED DYNAMIC SLICE BACKSOLVING

By

CLARK WOOD

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Masters of Science

Degree Awarded:
Spring Semester, 2014

Clark Wood defended this thesis on April 19, 2014.
The members of the supervisory committee were:

Zhi Wang

Professor Directing Thesis

The Graduate School has verified and approved the above-named committee members, and certifies that the thesis has been approved in accordance with university requirements.

ACKNOWLEDGMENTS

SFS, for the generous funding, mentoring, networking opportunities which have given me the opportunity to study computer science. Specifically my co-principal investigators, Mike Burmester and Xiuwen Liu.

Jonathan Salwan, for his pintool code examples.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Finding software bugs is a difficult but lucrative endeavour, and fuzzing is one technique often employed for finding bugs, particularly exploitable vulnerabilities. However, when no source code is available, fuzzing complex programs effectively can prove difficult, and as techniques become more sophisticated they require more resources and time. We present a technique, automated dynamic slice backsolving, which, to the best of our knowledge, represents a novel way to improve fuzzers' code coverage, without resorting to heavyweight techniques like symbolic execution. We demonstrate this technique with a proof of concept fuzzer which explores paths in a binary to effectively reveal a secret, performing an oracle attack without an oracle. Our approach requires no source code, no manual instrumentation, and no symbolic execution, and, as our results show, can reliably discover and explore new execution paths in a timely fashion. We believe dynamic slice backsolving represents a potentially strong complementary technique for modern fuzzers to implement. In addition, we plan to release and continue development on Stubble's DTA framework, since we not aware of many freely available and active projects.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Fuzzing, or the feeding of manipulated input to processes to try and detect software bugs and vulnerabilities, is an integral, even required, part of the work conducted by both private-sector companies like Microsoft [Q] and Google [AM] and nation-state powers and contractors. In 2014, Pwn2Own paid out 850,000 USD to participants over two days for proof of zero-day exploitable vulnerabilities, for an average of 66,000 USD per exploit [AN]. Forbes suggests black market prices are considerably higher, peaking at over $150,000 for modern browser exploits [AO].

Because of the high market value of finding bugs, either to patch or exploit, many researchers create their own fuzzers, or use open source products like Peach [Z]. These fuzzers vary in purpose and sophistication. The simplest of fuzzers may have little or no understanding of the program it intends to fuzz, merely changing random bytes in user input and monitoring for interesting situations which hint at bugs, such as segmentation faults. A log is produced, which can be analyzed by a programmer at a later date to try and identify the source of the crash. However, fuzzers of this variety, while adept at causing program crashes, tend to find shallow bugs or exit conditions. This is because, with no understanding of the inner workings of a program, a fuzzer is likely to fail checks at the start of a program over and over again [T]. As an example, fuzzing an application which implements HTML by mutating random input bytes will have to reproduce the complex markup tags required to produce valid HTML. It will expend valuable time and resources failing to create valid head and body tags, instead of finding deep bugs.

In order to fuzz more smartly, researchers employ several techniques, two of which are significant for this paper. Dynamic taint analysis, or DTA, provides valuable information for guided fuzzing. DTA is a form of dynamic analysis, which marks user input as tainted at some level of granularity, and follows this taint as it propagates through the binary. Certain, task-dependent sinks are identified, and if taint flows to one of these sinks, a trace of program execution allows analysts to follow how taint sources are linked to these sinks. Symbolic execution first runs a program concretely to gather constraints on a particular element, then executes the program symbolically, giving these constraints over to some kind of constraint solver which decides what values the element should given the recorded conditions. Symbolic execution improves code coverage, helping to guide fuzzers into deeper, more interesting parts of a program. Symbolic execution, however, is computationally expensive, with naive implementations running in time exponential to the number of branches [C], and often requires source code availability and modification, which leaves room for refinement. Our technique uses DTA to hopefully achieve greater code coverage than random testing, without having to convert code to an intermediate representation or outsource constraints to a solver like symbolic execution.

1.2 Problem Statement

The software community knows that programmers write buggy software, even with debugging tools and access to source code for auditing purposes. Fuzzing, however, cannot even assume source code access, because interesting programs are not always open source. As an example, Internet Explorer, a prime exploitation target, is propietary software. Successful testing depends upon the fuzzer's ability to explore the binary as a black box and detect bugs without source code. Reasoning about the inner working of a complex application when one can only observe the relationship between input and output is hard. In addition, each branch instruction in assembly language introduces two new possible paths, leading to an explosion of explorable paths as binaries grow in size and complexity. Following all or most these paths, and doing so in a timely manner, requires complex software and time-memory tradeoffs. These two difficulties are called the code coverage problem and the path explosion problem [U] respectively. Many current fuzzers use techniques like dynamic taint analysis and symbolic execution to improve code coverage, and often employ various heuristics to mitigate the path explosion problem.

Our work aims to address the code coverage problem in a similar way to symbolic execution, while sidestepping or avoiding entirely some of the negatives associated with symbolic execution. In particular, symbolic execution traditionally:

- Requires source code access
- Requires programmers to instrument code manually

- Must convert code to an intermediate representation
- Outsources constraint solving to heavyweight satisfiability solvers

Because of these downsides, much research tries to attain more reasonable performance by using symbolic execution less or more wisely [J]. We focus instead on approximating symbolic execution's code coverage by using dynamic taint analysis to drive test case mutation. As the native program executes, we dynamically instrument the binary and record execution slices. Memory accesses, loads and stores, and transformations of tainted bytes are recorded, and then used to backsolve to give exact byte values and locations to explore in future runs. Our implementation of this technique requires no source code, no modification to a binary, no conversion to an IR, and no SMT solvers.

1.3 Contribution

In this paper, we present our analysis of existing techniques along with our own contributions and conclusions. We begin by exploring related work in Chapter 2, which sets the background and describes current fuzzers' techniques and limitations. In chapter 3, we describe our contributions, in the form of what is, to the best of our knowledge, a new technique for black box binary fuzzing, a proof of concept fuzzer which implements the technique, and testing results. Lastly, we describe possible future work and conclusions from the experience in Chapter 4.

Our technique, automated dynamic slice backsolving, uses dynamic taint analysis to watch user input. As user input is introduced, each byte is tracked separately and, when the program transforms, uses, or moves user input, those actions are recorded. When branches which depend upon user input are reached, the recorded slice, if possible, are reversed, backsolving for a new value of a particular byte in user input which will lead execution down a different path in the binary. After execution finishes, input test cases are mutated to reflect this new information, and a new execution test runs.

The inspiration for this technique comes mainly from ”A Taint Based Approach for Smart Fuzzing”, which proposes a fuzzing tool architecture, involving vulnerability detection, followed by taint analysis based upon found vulnerabilities, which drives the generation of intelligent tests. Dynamic taint analysis tracks which bytes of user input are important in the binary, and these so-called ”hot bytes” are made the focus of fuzzing attempts. The paper mentions it is valuable to be able to generate backward slices of program traces. These traces can then be run and analyzed by programmers. In the spirit of automation, our work not only generates backward slices based on taint propagation, but also automatically solves these traces to find new byte values. Thus, for a subset of programs, we can determine not only the exact location of user input which determines a branch condition, but also generate precise values for both branch and non-branch cases.

We have implemented this technique in a proof of concept fuzzer, Stubble, which operates on x86 programs. Stubble dynamically instruments binaries, and automatically generates test files which exercise new paths in the input program. Program input, output, and interesting conditions like segmentation fault signals are recorded during execution, and taint propagation is recorded from user input sources to branch sinks. When a tainted branch is found, stubble automocically backsolves the execution slice and generates a new test case which will drive the program down the opposite path explored in the current execution.

To test Stubble's ability to dynamically discover new branches and generate test cases which will drive future runs down these new branches, we have designed several binaries which receive user input in the form of a file. These binaries open and read the file, comparing a subset of the file contents against a secret. If the file contains the secret, then a win condition is printed, otherwise a lose condition is printed. For the proof of concept, this win condition could be thought of as a segfault, or other interesting condition that suggests an exploitable condition exists. We show that stubble is able to discover all paths in the binary, particularly the interesting paths involving the win condition, without any prior knowledge of the secret. Stubble works similarly to an oracle attack, able to discover one character within the secret per run, and preserve work across runs to crack the secret in O(n) tests, where n is the secret length in characters. In contrast, random byte fuzzers, which exhaustively test all user input bytes which their model declares valid, run indefinitely and have no guarantee of discovering the win condition. We also present a key observation for improving path exploration in the x86 architecture, namely the finite number of conditional branch statements and instructions which modify the flags which branch statements use. Targetting these instructions specifically pays dividends. We believe automated dynamic slice backsolving is a reasonable complementary technique to dynamic taint analysis and symbolic execution for binary fuzzing, and that stubble provides a valuable proof of concept for future research into smart fuzzers.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Microsoft SAGE

Microsoft's Scalable Automatic Guided Execution, or SAGE, is an example of a white box fuzzer. Researchers at Microsoft developed SAGE to deal with large, complex programs which needed to be automatically tested for deep bugs. Microsoft considers SAGE, and whitebox fuzzing in general, so important that it requires the practice in its Security Development Lifecycle. The basic SAGE execution begins when a program to be fuzzed is first run concretely with known good input. During this run, the program is instrumented with Microsoft's AppVerifier to watch for bugs. Afterwards, SAGE receives a copy of the x86 instructions executed during this run, and symbolically executes the instructions, gathering constraints to be solved later by an SMT such as Z3. Once it has a set of these constraints, SAGE will negate each constraint in the set one by one and feed this new set to Z3. Sets that pass are ranked in descending order by the number of new instructions they reveal, and return into the start of the loop, to be executed concretely again [Q].

Because SAGE is propietary, the lack of detail about certain procedures raises questions. For instance, how does SAGE know when an instruction is new? There are a limited number of x86 instructions, which can be combined in an unlimited number of ways, which means it can be difficult to tell when a truly new instruction is executed, unless one keeps a directed cyclical graph of all explored instructions, and follows a run's progress down this graph during execution. In addition, problems integral to this work, such as code coverage and path explosion, must be addressed [Q].

SAGE uses symbolic execution, and prunes paths taken with a propietary ”generational search” algorithm. This algorithm prioritizes generating new test cases. It takes a set of all constraints and tries negating each constraint independently to generate a new test case. SAGE then symbolically re-executes the constraint set using a trace analysis engine [Q]. Similarly, s2e uses QEMU for instruction set emulation, and then appears to be a strong trend in white-box fuzzing.

In practice, because program termination is indeterminable, white box fuzzing could run forever. And since programs at Microsoft can easily exceed one million lines of code, with complex control flow structures, pointer manipulation, and other difficult to account for code constructs, SAGE employes several performance enhancing techniques to keep fuzzing practical and economical. Symbolically executed instructions and local constraints are cached, and unrelated constraint elimination rids the execution of contraints not related to the constraint under test. How this is determined is anyone's guess, since it is impossible to know which constraints are independent until one has explored all the code in a program. SAGE also employs a flip count limit, allowing only so many negations of a particular constraint [Q].

This is helpful for dealing with loops, which might evaluate or modify the same constraint many times. This technique is an example of a typical tradeoff for fuzzers. A lot of program execution time is spent in loops, but deep bugs can lurk in rarely exercised parts of a loop, so fuzzers must pick some heuristic for terminating the exploration of a loop, trading code coverage for running time.

2.2 Fuzzing Web Applications

Fuzzing is a reliable and inexpensive way to find vulnerabilities not only in compiled C/C++ programs, but also in higher level scripting languages. JavaScript, as an example, is a higher-level scripting language becoming more and more prevalent as we continue to connect ever more devices to the internet and grow individual people's connectedness. Web browsers, as a portal through which to interact with millions of users, are a common target for malware and other exploitation, and JavaScript, as the de facto client-side scripting language, offers many targets.

Web exploits such as cross site scripting (XSS) are possible because of language functionality like the eval method, which takes an arbitrary string as an input parameter and evaluates that string as if it were JavaScript source code [AE]. The authors of "A Symbolic Execution Framework for JavaScript" wrote Kudzu because of a perceived lack of sophisticated tools for fuzzing client-side Javascript. In particular, constraint solvers at the time seemed unable to reason about JavaScript string objects, which are used in various parts of the web to represent myriad types of data, and are then parsed by custom, open-source, and propietary libraries in order to cast from a string to another data type. As an example, consider the following snippet involving eval:

```javascript
// Let userInput = "2 + 2"
var str = eval(userInput);

```

```
if (str == 4) {
        everythingDies();
        throw new Error;
}
```

In this contrived example, catastrophe occurs when user input, when executed as JavaScript, returns the value 4. Without even considering the strange casting and equality rules which JavaScript operates under, there are multiple ways for user input to throw an error, making constraint solving particularly difficult. The case where userInput = ”4” is straightforward, but other cases which do not look like the String ”4” or the Number 4, but evaluate to it, can also trigger the error. According to the authors, before Kudzu, string solvers were unable to reason about regular expressions, string equality, or multiple variables [AD].

While JavaScript strings, and their ubiquitous usage in client-side web programming, presented one obstacle, the sheer complexity of modern web browsers presented another. Typical browsers can open and display PDFs, play media files, and execute multiple markup and programming languages like HTML, CSS, and JavaScript to present a web page to a user. Because of this complexity, the authors decided to divide possible user interaction into two, disparate spaces: an event space and a value space. The event space contains HTML elements and their state as defined by user behavior. This would include HTML elements like radio buttons or text boxes, along with their current state, such as which radio button is currently selected, along with the state transitions for the current page, such as a user clicking one radio button, then deselecting to check boxes in order. As a rule of thumb, GUI elements which users can interact with reside in the event space. The notable exception to this is any GUI element a user types into, such as a text box. Value space contains the values for these GUI elements, along with any other text which is used as input which users or other web pages have control over [AD].

Kudzu aims to discover vulnerabilities, particularly client-side code injections, by using dynamic symbolic execution to identify constraints placed upon particular input and then solve them to generate new test cases. Dynamic symbolic execution, which mixes concrete values with choice symbolic values to optimize for speed and code coverage, is basically concolic execution. Kudzu was developed for the WebKit browser and consists of two major parts:

1. GUI explorer

2. Dynamic symbolic execution engine, which includes:

(a) a constraint extractor

(b) a constraint solver

The GUI explorer handles the event space by exploring all possible sequences of user-defined events at random. In order to discover event handlers, the GUI explorer relies on information from Kudzu, which instruments WebKit functions. When an HTML element which can generate an event is found, or when one of these elements in removed, Kudzu's hooks notice this and inform the GUI explorer accordingly. Symbolic execution is broken down into a custom constraint extractor, which acts upon an execution trace from a concrete execution of the program, looking for control-flow statements and whether they were taken given the appropriate values compared, and a custom constraint solver, purposefully designed to handle JavaScript objects, which receives symbolic constraints from the extractor. After the solver has reasoned about the constraints on some piece of input, a new input test case is generated and fed back into the system. Since symbolic execution is costly, Kudzu identifies when user input appears to flow to one of finitely many vulnerability sinks, such as eval or document.write. Only this user input is considered symbolically. As Kudzu runs, it takes new test cases and executes the program with a mix of symbolic and concrete variables, and checks to see whether input that flows to vulnerable sinks can result in an exploit, for instance by inserting JavaScript alerts and checking to see whether they execute [AD].

Similar to SimpIL [C], Kudzu tries to circumvent the issue of solving complicated constraints by first converting JavaScript into an intermediate language called JASIL. Strings, in particular, are parsed and partitioned into several sub types which are more uniform. For instance, one type handles strings which are semantically regular expressions, another handles strings which are the concatenation of two other strings. Although not as featureful as JavaScript strings, this IL captures a large enough number of String features to allow Kudzu to find previous unknown exploits in programs. Shortcomings include an incomplete support of String operations and JavaScript regular expressions, notably backreferences, which match a previously matched string in the regex. Although representation of strings is incomplete, Kudzu reasons well enough to find two new vulnerabilities, as well as 9 previous vulnerabilities [AD].

2.3 Taint-based Fuzzing

The major impetus for this work is outlined in "A Taint Based Approach for Smart Fuzzing". This paper proposes a fuzzing tool architecture, involving vulnerability detection, followed by taint analysis based upon found vulnerabilities, which drives the generation of intelligent tests. As tests are run they are checked to ensure they provide adequate code coverage, and results are carefully monitored for interesting situations like crashes [A]. The paper also serves as an excellent explanation of the current state and future of fuzzing. In particular, they suggest combining taint analysis with backward slicing. The technique discussed in this paper boils down to an attempt to automatically solve slices like those introduced in [AA], which have been derived from taint analysis as much as possible.

2.4 TODO Mayhem, AEG, Exploitable Bugs

While finding bugs of any sort enhances software by providing developers the opportunity to fix problems with their programs, finding bugs that lead to exploitable conditions are of special interest. A bug is exploitable if it allows adveraries to undermine confidentiality, integrity, or availability via avenues such as information disclosure, denial of service, or remote execution of code. Because these bugs stand alone as a separate, particularly worrying class, efforts have been made to automate the process of proving whether an identified bug is also a vulnerability which can be exploited.

Mayhem, an exploitable bug-finding system, is an attempt to address this issue by exploring state space as efficiently as possible. The authors designed Mayhem to handle path explosion by monitoring system resources and balancing the need for concurrent exploration with the resources consumed by forking and running many processes. Mayhem accomplishes through hybrid symbolic execution, a technique which heuristically combines offline and online symbolic execution. Offline symbolic execution first runs a program concretely, instrumenting the process to gather information which will then be used by a separate symbolic execution engine. Online symbolic execution, instead, tries to explore the entire program symbolically by forking new processes at branches. [H]

Mayhem improves fuzzing time performance by weaving between these two modes. It is implemented as a client-server architecture, with the client running code concretely and passing specific blocks to the server for symbolic execution. Only the server outputs results or further test cases to explore concretely. The client uses block-level dynamic taint analysis, passing control to the symbolic execution server when tainted branches or jumps are discovered. Mayhem modifies its behavior based on system memory constraints, beginning in online mode and switching to offline mode when a threshhold for memory resources is reached. Upon switching, Mayhem also saves checkpoints to save its work in an attempt to reduce exploring the same code twice. [H]

AEG required source code, where Mayhem needs access only a binary to execute. Mayhem is, however, limited to a subset of all system calls for the operating system it executes on. This is deliberate on the part of the authors, who reason that OS-wide symbolic execution introduces too much overhead. Of significance as an area of future research, Mayhem, and most other fuzzing technology explored, cannot reason about taint propagated through threading mechanisms [H].

implications, but mine doesn't require a patch?

2.5 Checksum-Aware Fuzzing

Much of smart fuzzing involves reasoning about constraints gathered during the execution of instrumented code. TaintScope [AB] was developed as the result of the following observation: generation-based fuzzing improves as the quality (and consequently effort) of the model to fuzz improves. This means better fuzzing is accomplished by devoting more programmer time and effort, instead of allocating more computational resources. Since a programmer's time and mental resources are valuable, so fuzzers should do for them by yielding fewer test cases that result in more interesting crashes [T].

TaintScope focuses on checksums, one where area where traditional symbolic execution-based fuzzers tend to perform poorly. Other areas, which could be interesting areas of further study, are essentially any other functions that are, in one set of assumptions or another, considered one-way. For instance, bitwise shifts, being lossy, are hard to execute symbolically, as are encryption and decryption or hashing without a precomputed table. The impetus for TaintScope's design was the desire to find deeper bugs in work like network protocols, which very early on compute checksums for integrity. Before TaintScope, fuzzing said protocols often lead to lots of worthless test cases, because they all lead to similar, shallow crashes involving failing the checksum test. Since there are far more ways to fail a checksum than to pass it, fuzzers that couldn't reason about the checksum or threw random bytes in the checksum field wasted a lot of work. Instead, TaintScope identifies likely checksum integrity checks and instruments programs to let them keep going to try and find deeper, more meaningful bugs [T].

The general method is to:

1. Perform dynamic taint analysis during concrete execution
2. Identify what could be checksum validations
3. Run the instrumented program again, modifying ZF in EFLAGS to always "pass" the checksum
4. If a crash results, execute again with the checksum bytes marked symbolic

In order to identify possible checksum validations, TaintScope assumes certain x86 branches exist to validate the results of computing checksums. With correct input, the checks leading up to these branches are always either true or false, and with incorrect input they are always the opposite. During execution TaintScope watches for these conditions, and once these conditions, and the bytes involved in them, are identified, it treats the bytes and symbolic, collecting constraints during execution. After this, TaintScope can alter the previous test case to pass the checksum. However to save work, it only spends time solving these bytes when the normal test case results in a crash. To get past shallow crashes where checksums fail it will merely hot-patch the ZF before identified "checksum-based jumps". Note this approach won't work with hashes, since one-way functions can't be easily solved with constraint solvers [T].

This general scheme is accomplished with fine-grained dynamic taint analysis, that is, taint analysis at the byte level, as opposed to a more general boolean level of taint, where perhaps user input is tainted and all non-user input is untainted. As fuzzers evolve they appear to favor more and more finely-grained taint analysis.

TaintScope implements DTA using PIN to instrument syscalls in binaries. PIN_AddSyscallEntryFunction and PIN_AddSyscallExitFunction are used to monitor syscalls which are known to introduce taint to a system, such as open, which might open a user-specified file, or read, which might read user-defined bytes into memory. Taint is followed as it propogates throughout data via move instructions and arithmetic instructions. TaintScope's tainting policy however, transfers taint by unioning taint values whenver they both appear in an instruction. Thus, if the policy is enforced strictly, and eax and ebx were both tainted, mov eax, ebx would regard eax as now tainted with both eax and ebx, when in reality the value of ebx has been moved into eax, removing the original taint from eax. This is one example of overtainting, which should be taken into account to achieve acceptable performance. Note also that, as of the paper, TaintScope does not take into account control-flow dependencies when tracking taint [T].

DTA is used to identify the so-called "hot bytes", which are then used as symbolic values later on, should test cases warrant further attention because of a crash or other interesting behavior. This strategy has netted TaintScope 27 new vulnerabilities across several different pieces of software, most of which operate on complicated file format specifications like PNG or PCAP files [T].

2.6 Intel Pin

Heavyweight analysis techniques, like symbolic execution or dynamic taint analysis, which operate over an entire instruction set, must account for the idiosyncracies and diverse operations supported by the architecture. By leveraging a pre-existing framework like Intel Pin [M], new software can abstract away from architecture specific details like x86 variable length instructions and avoid re-inventing the wheel.

Pin, developed by Intel Corporation and the University of Colorado, performs dynamic binary instrumentation on x86, Itanium, and ARM architectures. Pin takes an arbitrary binary to be instrumented and a user-developed dynamically linked library, and instruments the binary at runtime by inserting user defined routines or changes into the application's code.

Pin's architecture is two-fold: consisting of a virtual machine for compiling and executing code, and a code cache for storing invididual code traces. An API for developing Pintools is also provided. Pin takes apart the program one trace at a time, compiling the native executable and the instrumentation code together into the same architecture. Pin defines a trace as a linear sequence of machine code instructions, terminated by either an unconditional jump or a pre-defined number of conditional jumps. Traces, hopefully, correspond to higher level control flow structures, but are also limited in size by a pre-defined maximum number of instructions [M].

The Pin VM can be broken down into three sub-components: a just-in-time compiler which instruments the binary by hooking the appropriate routines and re-compiles into the original architecture, along with a dispatcher and an emulator. The dispatcher passes code to the code cache and executes instructions, but, because Pin resides solely in userland, certain activities which occur at a lower level of the operating system, like syscalls, cannot be captured normally. The emulator handles these cases [M].

Pin injects itself into a given binary using Ptrace to modify instructions and then trap running processes to inject code in a similar manner to how debuggers like GDB attach to processes and inject software breakpoints [AK]. Just like GDB, Pin can attach to a running process or instrument a process from the first instruction. This stands in contrast to other dynamic binary instrumentation frameworks like DynamoRIO [AL], which use LD_PRELOAD to force a new shared library into the process context. This noisily shifts shared libraries' memory addresses to accomodate the new, pre-loaded library, and also cannot instrument the first parts of a program before the library loading code executes [M].

Pin instruments at the trace level, JIT compiling a trace and caching it for re-use. A number of Pin optimizations revolve around the compilation phase, because testing shows Pin bottlenecks tend to concentrate around the execution of instrumented code. This is perhaps because instrumenting code involves hooking perhaps both the entrance and exit to a function, which adds two jumps. In addition, transitions between the VM and code cache require saving application state. The eflags register, for instance, must be pushed onto the stack to be read or written so that the program's original eflags is preserved. Since this register is used for all jumps, among other things, it is accessed often by both Pin and the instrumented program. Thus, Pin analyzes the liveness of each flag in eflags on a per-routine basis, and avoids saving and restoring flags which are not used by the routine as an optimization measure [M].

This research leverages Pin to instrument arbitrary binaries at the instruction and syscall levels. Code is inserted to track taint introduced by user input, and recorded separately as it propagates. When taint reaches a branch, the recorded state information is used to work backwards through lossless operations to try and reason about the location and value of bytes which affect whether the branch is taken or not.

2.7 Techniques

Several valuable techniques for improving fuzzers' code coverage have evolved over the years. Two, in particular, inspire this work directly: Dynamic Taint Analysis and Symbolic Execution. Dynamic Taint Analysis, or DTA, provides valuable information for guided fuzzing. Symbolic execution improves code coverage by first running a program concretely to gather constraints on a particular element, then executes the program symbolically, giving these constraints over to some kind of SMT solver [CITE] to solve for what value the element should have under certain conditions. Concolic execution, which has spawned out of the desire to use symbolic execution more wisely due to performance constraints, aims for reasonable runtimes while maintaining good code coverage. These each come with their own drawbacks, a primary one of which is the lack of automated instrumentation. Historically, any amount of symbolic execution required a programmer to directly instrument source code, leading much current research to focus on automating as much as possible.

2.7.1 Dynamic Taint Analysis

DTA is a form of dynamic analysis, meaning analysis occurs during execution of a program. This can be contrasted with static analysis, where a program's source code is analyzed without executing the code. The IDA Pro Disassemble [AH], which takes a compiled C binary as input and statically disassembles it to produce some flavor of assembly code, is an example of a static program analysis tool, whereas a debugger, which instruments and executes a program, allowing users to manipulate and watch the process at runtime, is an example of a dynamic analysis tool.

Developers often have an incentive to obfuscate payloads or manipulate code to make static analysis more costly, such as malware and exploit writers who don't want their products detected and reverse engineered. This merely raises the bar for attackers, since software cannot be altered to the point where it doesn't perform its original function merely to escape detection and analysis. As an example, a program that begins life encrypted must at some point use or download a secret key to decrypt itself and run. This might be incredibly difficult to find using only static analysis, but by running the program and letting it decrypt itself, then snapshotting the program's state, we can circumvent the protection entirely. Thus, program obfuscation, reverse engineering and anti-RE are cat-and-mouse games. This is why dynamic analysis is so important, and it is an open and actively researched question whether there even exists a way to provably make a program strongly obscure [AI].

In [C], the authors summarize dynamic taint analysis and symbolic execution and attempt to formalize each technique. A formal definition for DTA requires a formal language upon which to perform taint analysis, so the authors use SimpIL, a Simple Intermediate Language. Note that this is distinct from but related to the intermediate representations which some fuzzing research converts sources into. When fuzzers first convert to an intermediate language, this is done to simplify future analysis, as when Kudzu [AD] converts JavaScript strings into one of several new data types which are easier to reason about. Time is traded up front in order to make future operations easier or possible. SimpIL is a complete programming language, lacking in features but used as a base for some of the author's other work. It should be noted that the features SimpIL lacks, in particular functions and variable scoping, can be implemented using SimpIL, and SimpIL expressions not having side effects similarly do not hinder it. The important implementation issues for DTA spring from one's tainting policy. Our tainting policy decides under what circumstances we introduce taint, verify it, and track its propagation throughout the program execution [C]. Introduction rules should specify:

- when taint first appears in a program,
- what type of taint it is, and
- what parts of the program have become tainted as a result

We can record taint at varying levels of granularity, with the tradeoff of finer-grained taint requiring greater system resources and slowing down analysis. Different types of taint may be more interesting to different kinds of fuzzers because different input comes from sources of varying trustedness [C]. For instance, we should probably trust raw, unsanitized user input we have read less than we trust a configuration file that only privileged users can modify. Both might be read in as input to a program, but the former is much more likely to contain malicious data like shellcode trying to exploit a vulnerability. Fuzzers have continued to drill down into finer and finer levels of granularity, from assigning everything with a generic binary value of tainted/not tainted to byte and even bit-level marking of which part of user input affected which variable. [CITE] [C] Note that this also expands taint's data type from a bit to something larger, and each tainted object requires this tag.

Just as with taint introduction, we verify taint based on our overall objective. If we are worried about a binary being exploited to execute arbitrary code, we may instrument the EIP register. When we verify that taint has reached EIP, we know we stand a strong chance of having found an exploitable condition. Likewise, were we monitoring taint not in order to generate test cases for fuzzing, but to study a piece of malware, we might instead watch for when a particular routine, which we think is unpacking the malware, has finished, and the malware is now available in an unobfuscated form.

Taint propagates through a program from its initial source, perhaps making its way to vulnerable sinks. The condition we are interested in is when user input finds its way to these vulnerable sinks in a program, and is able to influence them. As an example, user input received on STDIN which is able to change the value of the EIP register in x86 assembly is able to hijack control flow in the program. This may be as innocuous, though annoying, as forcing EIP to strange areas in the binary, resulting in segfaults, to more sinister remote code execution via pointing EIP to user controlled shellcode.

There are several challenges involved with implementing DTA. First of all, accurate taint analysis, as free as possible from both over and under tainting, requires a precisely implemented taint policy. Different types of control flows, either direct or indirect, must be handled as well, and one-way or lossy functions also tend to be either difficult or untenable to instrument.

[C] suggests that taint introduction is an easier problem than taint removal. After all, taint is introduced into a system by a limited number of avenues, such as files, user input, and the execution environment. Once these are instrumented, they can be considered as always introducing taint to the system. However, taint removal depends not only on the instruction execution, but also on the context of the instruction. In x86, performing the exclusive or operation with the same register as both operands will zero out the register. Thus, even if the eax register were tainted, after xor eax, eax the eax register will be untainted, as its value is always 0. Taint policy could, for instance, add a rule that xoring a register with itself will remove taint from the register. However, what if eax and ebx are both tainted with the same value and we execute xor eax, ebx? Now the check must not just blindly remove taint when the same register is both operands, but instead consider the source of taint found in both operands. Failing to account for circumstances like this introduces overtaint, which means we spend time analyzing false positives.

Even worse is undertainting. Overtainting creates unnecessary work, but undertainting may lead us to miss taint propagations to vulnerable sinks. Undertainting typically occurs when we mishandle or fail to perceive information flows [C]. Taint can propagate via explicit or implicit flows, sometimes also called data and control dependencies respectively [G]. Explicit flows involve a tainted variable, x, which is used in an assignment expression to compute a new variable, y. In this situation, x taints y, and if y is involved in any further assignment to a variable z, then x taints z by transitivity. In contrast, implicit data flows involve a tainted variable used to affect control flow within a program which subsequently sets the value of another variable, for instance at a branch [D].

Handling taint correctly is of great importance because of the existence of subtle implicit flow cases referenced by Clause, J. et al [D]. Implicit data flows are not always considered by dynamic tainting techniques [E], but for our project, implicit data flows are very important, so we need to implement DTA which catches both implicit and explicit flows.

DTA has been used to implement smart mutation fuzzers [A, T, MORE], but we believe the technique we explore later is unique from and complementary to Bekrar's work. In [A], DTA is proposed as a way to intelligently decide which parts of user input should be mutated. In [T], dynamic taint analysis identified hot bytes, which were then modified randomly or with boundary values. We plan to add on to these ideas by allowing smarter mutations to be selected by backsolving to decide which values are most likely to result in new paths being explored. This has been done with symbolic execution before, but traditional symbolic execution is much more heavy-weight, both in time required to setup the environment and in overhead introduced to run the program, than our proposed approach.

2.7.2 Symbolic Execution

Symbolic execution supplies symbolic instead of concrete values for input [O]. This technique has been used to effectively detect bugs in software [Q], although historically it has required access to source code, which makes traditional symbolic execution infeasible in many vulnerability research situations where source code is unavailable. In addition, symbolic execution often uses instruction translators like QEMU [K] and satisfiability modulo theorem solvers like Z3 [P], which introduce significant overhead. This makes it difficult for symbolic execution to scale past the order of tens of thousands of lines of code.

Symbolic execution is defined under SimpIL as execution under which return values from input sources are symbolic, instead of concrete. As the program executes, we pick up various constraints on these symbols from expressions and branches. For instance, we may start with a symbol s, which is then multiplied by a scalar 10, and then added to a constant 5 to calculate the offset for an effective address in an array. The symbol s is now constrained as s*10+5. When we encounter branch instructions, we inherit constraints based on whether or not the branch is taken. If we branch in the figure below, we now have the constraints x * 10 + 5 == 25 [C].

```
s = get_input();
index = s * 10 + 5;
if A[index] == 25
  func1();
else
  func2();
```

Because symbolic execution can reason about the constraints placed on input, it drastically reduces the amount of testing necessary to explore all possible branches [C]. With purely random testing in the above example, we would need to provide many different values for s to trigger the if statement and explore func1. If fact, only one value, s = 2, leads us down this path, where all other values jump to func2. Just for integer based evaluations, we would need to test half of all integer values to stand a reasonable chance at triggering both paths of any if statement. Symbolic execution, in contrast, solves for s = 2 and s $\neq$ 2, requiring only two separate executions to explore both paths. However, this precision comes at a cost. Symbolic execution doubles the number of paths which must be explored each time we encounter a branch, since we can either take or not take each branch. Most optimizations for symbolic execution boil down to using symbolic execution less, parallelzing symbolic execution, or trying to simplify constraint formulas to make them easier to reason about [C]. Using symbolic execution less or more smartly falls into the realm of concolic and mixed execution. Parallelization introduces its own set of challenges, raised in [H], such as efficiently using resources by controlling the number of forked programs executing.

Not only is symbolic execution costly, but certain things are difficult to reason about symbolically. Accessing memory, for example, is hard to do when we reference not a memory location but an unevaluated expression. Solutions to this are imperfect and tend to rely on a combination of outsourcing the work to an SMT solver and performing static analysis on the binary to try and guess about memory locations. Schwartz, Avgerinos and Brumley raise the concern that malware could take advantage of most symbolic execution engines inability to perfectly reason about symbolic memory to frustrate malware analysts [C].

Certain constructs, like switch statements, when translated into assembly, are converted into jump tables, which consist of a sequence of checks and jumps to implement each case. During symbolic execution these compares and jumps could also be an expression instead of a concrete value. Syscalls and operations involving I/O are also difficult to express in a purely symbolic execution, because they can have side effects. For example, the write syscall not only returns an integer value representing success and number of bytes written, but also writes to a file descriptor. Implementation can model this effect to an external entity, but because of the unique side effects each syscall have, there will be many one-offs [C].

Without utilizing symbolic execution, however, it is difficult to automatically guarantee that all possible paths in a binary have been explored. Where most current research focuses on improving speed by using symbolic execution less, we propose to achieve similar code coverage to symbolic execution by implementing something akin to symbolic execution, but without using the SMT solvers and instruction emulation which tend to make symbolic execution slow.

Call-chain-backward symbolic execution has been proposed by [R], although this technique achieved a backward symbolic execution by iteratively applying a forward execution from successively farther away points in the program and the reducing the set of possible symbolic inputs. In contrast, we propose to avoid symbolic execution, forward or backward, by backsolving concretely and then applying heuristics to generate concrete input which is highly likely to result in exploring a new path. Symbolic execution never occurs, although we believe in many situations an equal degree of precision in determining new paths can be attained.

2.7.3 Mixed and Concolic Execution

TODO TODO TODO [C] considers concolic execution to be when we first execute concretely and THEN execute the same thing symbolically. Mixed execution is when we execute once with both concrete and symbolic input–make sure other works stand by this distinction.

TODO talk about how this is called many different things, and is a rather amorphous term.

Current research focuses on carefully deciding when and how to use symbolic execution [J, T, W]. So-called concolic execution mixes concrete and symbolic input, and various heuristics to determine when to resort to symbolic execution are in research.

Concolic testing has been shown to improve runtime while still allowing both wide and deep inspection of a program's execution tree, although it often still requires source code to be instrumented [X, Y]. What We propose is similar to concolic testing, in that we are augmenting concrete execution with dynamic taint analysis to solve for mutations which will exercise new paths in the code. However, the lack of a heavyweight constraint solver and the ability to work on binaries without instrumenting or using source code in any way have, to the best of our knowledge, not yet been proposed.

Koushik Sen defines concolic testing as combining ”random tesing and symbolic execution to partly remove the limitations of random testing and symbolic execution based testing” [AB]. This differs from hybrid symbolic execution engines like Mayhem [H], which switch between different types of symbolic execution. Hybrid symbolic execution does not itself supplement fuzzing by adding concrete random values. Both imply a tradeoff between two types of analysis, however, and thus both, when implemented by different people, will differ in where they assign the tradeoff. Some concolic testing, for instance, will start with concrete values which are random, or NULLs for pointer values. After an execution, one of the symbolic constraints will be negated, to allow a new path to be explored, and the program will be run again with new random input [AB]. Various research aims to either use symbolic execution less [Q, R, X], pick better than random values [A], or pick better input bytes to give random values [A, E, T, AB].

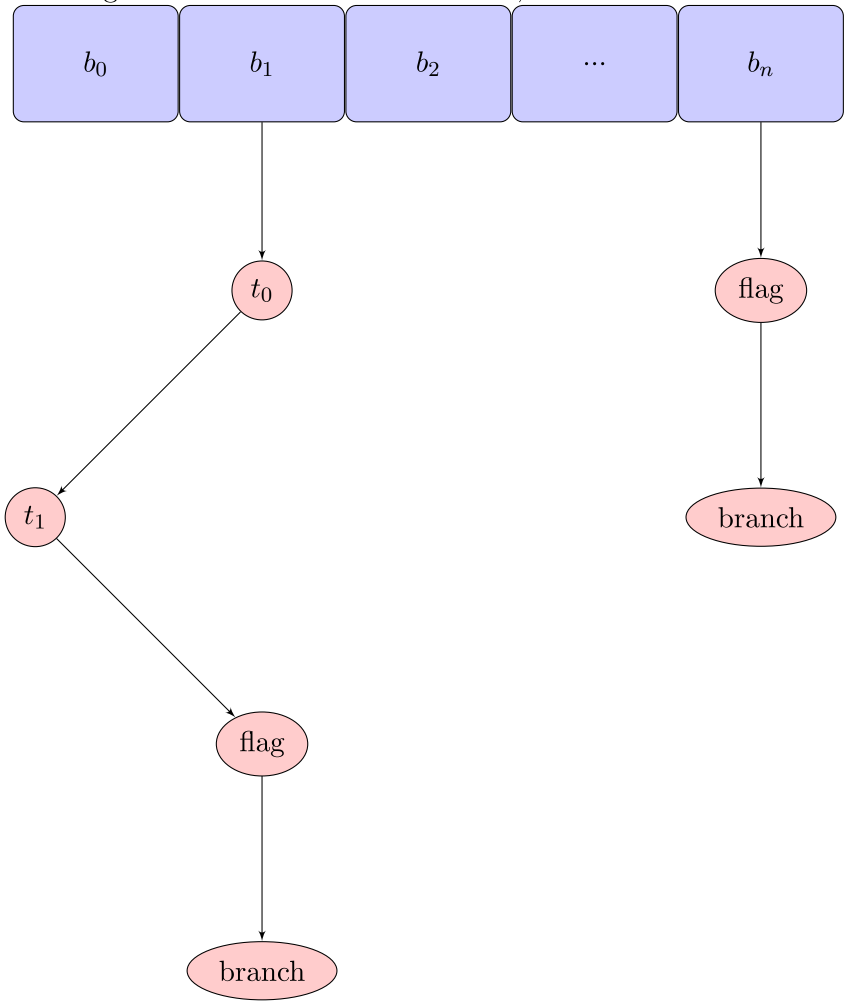
CHAPTER 3

CONTRIBUTIONS

3.1 Automated Dynamic Slice Backsolving

We have developed a technique which, to the best of our knowledge, provides similar code coverage to symbolic execution by using dynamic taint analysis to record relevant program slices and drive test case mutation. As a program runs, the introduction and propagation of taint is recorded. Should taint flow to a sink of interest, the recorded slice is backsolved to find new byte values which will hopefully lead future program runs down new paths. Taint is generally introduced to the program by user input. This can come in the form of user-provided files, command line arguments, or stdin. When user input is detected, it is marked as tainted at the byte level. This means that if user input is conceptualized as a stream of bytes $b_0, b_1, ..., b_n$, then the first byte is considered separate from the second byte and so on, with each byte individually followed throughout program execution. This level of granularity introduces significantly more overhead than binary tainting, which marks bytes as either tainted or not tainted, but provides far greater detail into which bytes of user input affect which parts of a program.

Once user input is captured and marked tainted, any memory accesses, loads and stores, and transformations of tainted bytes are recorded as they are encountered. This creates a slice of operations involving the tainted bytes, which can be followed like a trail of breadcrumbs back to the original position of the byte. This continues as the program executes until a conditional branch is reached. When a conditional branch instruction is reached, it is either taken or not taken, based upon the value of some comparison. If the branch condition depends upon tainted values, which have descended in one way or another from user input, then there is a strong possibility that the "road not taken" could be explored if only user input looked appropriate. In the case where all transformations to user input are reversible, we can backsolve these transformations and, based upon the comparison, modify the byte value so that it leads to the complement of the previous branch condition. Not only this, we can identify which byte $b_0, b_1, ..., b_n$ to modify, since we have followed each byte from their introduction. Using this information, we can mutate the previous test case to create a new test which will result in very similar program behavior, with the exception of branching where before it did not branch, and vice versa.



The idea for this technique is raised in [A], which suggests using dynamic taint analysis to identify "hot bytes" of user input, which affect important things in a program, and also to generate backward slices [AA]. Hot bytes become the target of fuzzing, with backward slices providing valuable context to analysts. Our contribution is the realization of this idea for a subset of cases, and the automation of solving these slices. Our hope is that by solving these slices automatically, we can identify not only the exact location but also the relevant values of hot bytes.

Our methodology focuses on following taint specifically up to tainted branches, to direct as much attention as possible to finding new paths to explore within the binary. This is of particular importance when source code is not available for analysis, which is common in vulnerability research. Compiled code can be disassembled or debugged, but compilation is a lossy process. Even industry de-compilers like Hex Rays cannot restore comments, since these are not preserved, and they cannot guarantee to reproduce the exact higher level language semantics, given only assembly language. Thus, our implementation relies on Intel Pin to dynamically disassemble compiled binaries.

Consider the following example in Intel x86 assembly:

```
call read             ;instrument read syscall
mov eax, [ebp - val]      ;taint propagates from val to eax reg
inc eax               ;transform eax
mul eax, 7            ;transform eax
cmp eax, 42           ;cmp and jmp based on tainted reg value
jnz 0x8BADC0DE
```

If we already had a test case where the value read onto the stack, after the transformations, did not equal 42, then the cmp instruction would not set the zero flag, or ZF, and we would jump to the memory location 0xBADC0DE. With dynamic taint analysis in place, we see taint introduced with the read syscall, then moved into the eax register (which is subsequently tainted). After an arbitrary number of transformations, perhaps zero, to the value in the eax register, comes a compare instruction, which checks eax's value against an immediate value. Then we can keep track of specific taint sources, and can record any movements or transformations, we have all the necessary information to ”solve” the branch. Once we reach the tainted branch, we know that user input can affect whether we jump or not, and consequently we can backsolve to find out how to affect the branch instruction. In particular, we know the conditional jump instruction jnz depends on the Z flag in the EFLAGS register. We record whatever instruction last modified each flag in EFLAGS, so we handle the compare instruction next. Because we have kept track of taint propagation, we know we can affect the value of eax, but not 42, since it is an immediate value, so we know we need to make the value in eax equal 42. We follow the instructions that have modified the value of eax and use any appropriate operands involved with the inverse operation of the one originally performed on the value to reverse the transformations to eax's value. Finally, we notice eax originally received taint when a value was moved from memory, which was read into memory using the read syscall.

We have implemented this technique in a proof of concept fuzzer, Stubble, which operates on x86 binaries. Stubble dynamically instruments binaries, and automatically generates test files which exercise new paths in the input program. Program input, output, and interesting conditions like segfaults are recorded during execution, and taint propagation is recorded from user input sources to branch sinks. When a tainted branch is found, Stubble automically backsolves the execution slice and generates a new test case which will drive the program down the opposite path of that explored in the current execution. Stubble is able to reason about large amounts of bytes of user input, for instance by performing oracle attacks on binaries which check user input against large secrets, requiring O(n) test cases to solve the secret with no previous information, where n is the length of the secret. Stubble has also been parallized at the process level, allowing for an arbitrary number of independent bytes to be reasoned about concurrently.

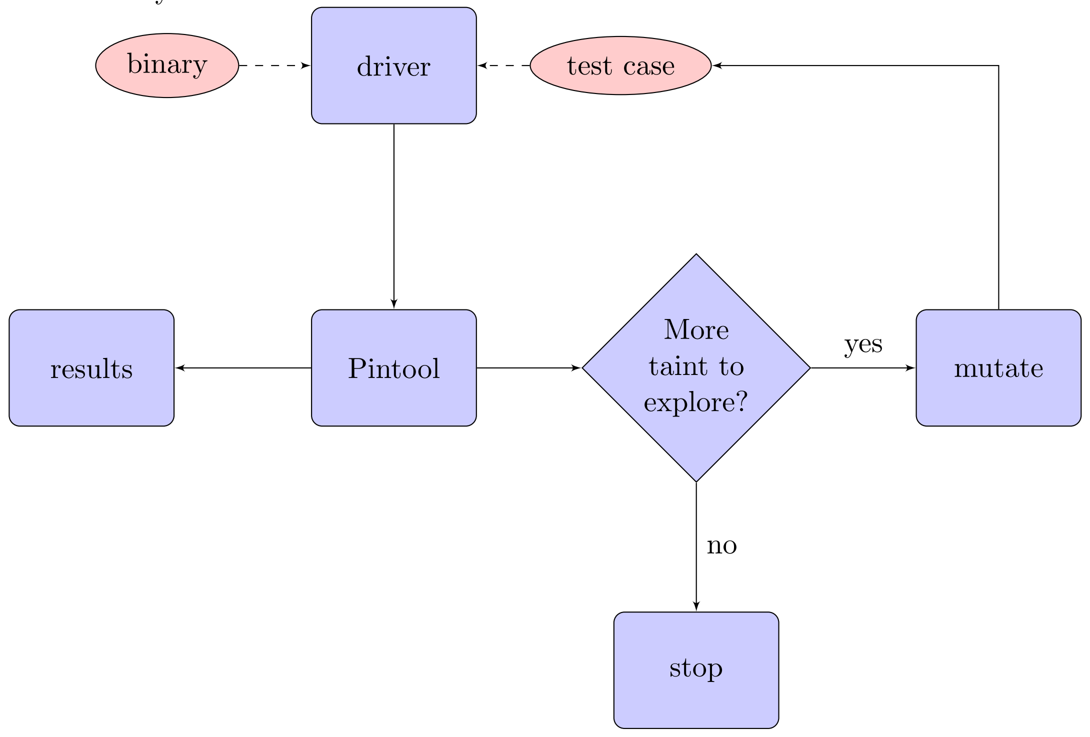
3.2 Design

TODO why intel pin and not dynamo rio, valgrind, etc.

Stubble consists of a pintool which implements DTA, and a Python driver which keeps track of test cases, input and output and segfaults recorded during execution, and employs dynamic slice backsolving to drive mutation using taint analyses provided by the DTA pintool. Pintools are arbitrary C++ which Pin compiles and inserts into a given binary during instrumentation. The driver accepts a valid x86 binary and a seed test case. From the seed test case, the driver runs the pintool on the binary with the seed test case as input. While the binary runs, the pintool notes when taint in introduced or removed from memory or from registers. If a memory location or a register is tainted, then any operations which move this data elsewhere or use this data in an operation spread taint to the appropriate areas. Moving untainted data into a previously tainted location removes taint from the system. For instance, if the eax register is tainted, then adding eax and ebx and storing the result in ebx will taint the ebx register. However, should an operation move the value of ebx to eax, then eax will no longer be tainted.

In addition, whenever stubble encounters an instruction that modifies the EFLAGS register, such as the cmp instruction, it is checked to see whether either of the operands is tainted. When this is the case, user input has some level of control over the EFLAGS register, so the appropriate flag in the register is marked tainted. When a conditional jump instruction is reached, the corresponding flag of which is tainted, stubble begins to backsolve with the goal of providing a test case to explore the other side of the branch. After an execution it publishes input, output, taint, and mutation data to the driver, which reasons about possible user input locations and values one byte at a time. This means when branch decisions are determined by comparisons at the byte level, the fuzzer can guarantee a new path is taken, provided it can accurately backsolve information about the transformations performed on the byte.

This is made possible by a key observation about the x86 architecture. There are a finite number of conditional branch instructions, which branch based upon the value of one or a combination of flags located in the EFLAGS register. In particular, the carry, overflow, parity, sign, and zero flags (CF, OF, PF, SF, ZF) are used to determine whether or not to branch. For example, the je, or jump if equal, instruction branches when the zero flag is set, while the jg, or jump if greater than, instruction branches when the zero flag is unset and the sign and overflow flags are equal. There are also a limited number of instructions which modify these five flags, and some instructions modify these flags unconditionally, but always setting, unsetting, or toggling them. Thus, we can monitor these flags in the EFLAGS register, and only must take note of the last tainted instruction that modified these flags, since untainted instructions cannot be controlled by user input and overwritten taint instructions cannot effect the branch instructions. This introduces little additional overhead, since we already must monitor each instruction executed to watch for track taint.



We assume ASLR is disabled for the binary to be fuzzed. This allows the program to easily track and compare execution paths. We have implemented the technique for the x86-32 architecture, because Intel Pin works on a few select architectures, of which x86 is the most prevalent. Since x86 is quite widespread, and vulnerabilities resulting from unsafe C code should persist across binaries, regardless of both the size and flavor of architecture (32 vs. 64-bit, ARM vs. x86 vs. PowerPC etc.), this should be no issue. Compilers try to adhere to the C standard and vulnerabilities like buffer overflows exist at a C source code level.

We also assume the presence of an algorithm which will detect an exploitable condition, provided it is given the proper binary and input to the binary, and an initial test case. For detecting real exploits, something like monitoring when tainted values overwrite the EIP register would be valuable. For the proof of concept we use a win condition hard coded into the input binary.

Fuzzers require intuitive but powerful test harnesses, so that users can easily control how a binary is fuzzed and understand the results of fuzz testing. Currently, stubble records input, output, and taint explored for each test executed, appending an auto-incrementing id to log files and separating results at the directory level. Stubble also keeps track of the mutations which have already been explored. Users can grep for interesting conditions like segfaults or unexpected output, then cross reference the test number with the observed taint flow, input file, and particular mutation tested.

3.3 Implementation

We implemented Stubble using Intel Pin. Pin instruments binaries by compiling a user-defined pintool and inserting the instrumentation code into the binary, converting directly from the target architecture to the same architecture, ex. from x86 to x86. The Pin API is very flexible, allowing for instrumentation at a binary-wide to an instruction-by-instruction level, and there exists an active community of developers. In particular, we base our work off of [N]. Because Stubble must reason about every taint propagation, we instrument at the instruction level, which introduces the most overhead. In addition to instrumenting each instruction, we also instrument the read system call. Taint is introduced when user input is read into memory. This section of memory, the contiguous memory addresses from read's buf parameter to buf + count, is marked tainted.

There are two places we consider taint to propagate to and from, namely registers and memory locations. We also must record operations that transform tainted data, so that Stubble can back-solve the transformations later. We record this data in data structures derived from the vector, list, tuple and unordered_map data structures from the C++ STL. The registers are represented as an enumerated array of tuples of (int, string). Registers start with tuples of (-1, ""), signifying that they are untainted. When registers become tainted, we insert values of (byteLocation, filename). Registers are altered as taint is overwritten or removed from the system, and since they only contain one value at a time, we mark the register as tainted by the specific byte in user input, using its location to refer to it. The EFLAGS register is a special case, since certain operations set specific flags, it is possible for one operation to taint the carry flag and for another to taint the overflow flag. Thus, we treat each of the important bits of EFLAGS as separate entries, with their own tuple devoted to them.

Memory is conceived of as an unordered_map of ints to (int, string) tuples. This is similar to the registers, except the key is the memory location. Since there are a small set of registers it makes sense to design an array which contains all of them, but many diverse memory locations may become tainted throughout execution, so we instead use an unordered_map. Memory is marked tainted or untainted similarly to registers. Lastly, we must record operations performed on tainted data. This is accomplished by representing user input as an array of lists. Each list entry is a string, marking one operation performed on the data. Stubble cross references each operation with the register and memory data structures to see which byte of user input should be updated with a transaction. After execution, a file of transactions for each byte of user input is printed, and the Python driver parses the operations recorded to backsolve new data values for each byte.

While finding and executing new code paths is valuable, a fuzzer without a vulnerability condition to monitor for cannot convey meaningful information to an analyst. There are many ways to detect the presence of a possibly exploitable bug, such as watching for tainted data reaching the instruction pointer, or watching program output for strings that suggest error conditions. Since program output is saved after each run, this is one possible way to detect vulnerabilities with Stubble, however, we specifically wanted to catch segmentation faults, since these are often raised when data corrupts pointers or overflows buffers or stack frames. Stubble monitors for segfaults using the Pin API. A signal-intercepting callback function for signal number 11, segmentation fault, is registered before execution is handed off to the fuzzed program. When a segfault occurs, information about the exception is made recorded and published in the signals/ directory.

To test Stubble's ability to perform an oracle attack, we prepared C binaries which checked user input against a secret. After compiling with the Tiny C Compiler, which performs minimal optimization and doesn't add security measures like stack cookies, this binary and a default test file are given as input to Stubble.

3.4 Challenges and Limitations

As a proof of concept, stubble lacks the featurefulness of a complete fuzzer. In particular, understanding more diverse types of user input, such as command line arguments, would prove valuable. Stubble currently only instruments the read syscall. When an appropriate input file is opened it marks all bytes read as tainted. A more featurefull fuzzer would instrument other system calls which could introduce taint, and would also handle C library functions which are the source of vulnerabilities, like unsafe string copying functions.

Note that dynamic slice backsolving requires transformations on tainted values to be easily reversible. Because of various cryptographic properties, encryption and hashing algorithms are not possible to backsolve. In fact, any lossy or not one-to-one operations prove difficult. As an example, the UDP checksum algorithm [AP] calculates the checksum of a packet by iterating over a pseudo-header, summing over each octets 16-bit 1's complement. Because of the repeating summing, an overflow is possible, and any overflow bits are added in later on. This means two non-identical packet headers could result in the same checksum, and so although we can backsolve to the original packet, we cannot make reasonable mutations which will result in the checksum we desire.

3.5 Testing

3.6 Results

To test stubble's ability to find new paths to explore in a given binary, programs were written in C, which opened a user-specified file and read several hundred bytes, then compared a subset of these bytes with a secret. The binary printed a success condition only if all bytes in user input were observed to be identical to its secret. Input to stubble consists of the binary in question and a seed input file. Input files were guaranteed to work appropriately (not crash the binary), but did not contain any signficant portion of the secret. Since the secret was multi-byte and random, it is possible that the input files contained small subsets of the secret.

This bears a strong resemblance to the classic oracle attack in cryptography. Consider that incorrect guesses drive execution down one of many losing paths. If an attacker can discover new paths, decide how to get to them, and continue to drive closer and closer to the goal in a timely fashion, then they are effectively exploring new paths in a similar way to how we want a fuzzer to act. Namely, they continuously find and explore new code in the executable, testing over and over for some condition of interest. Note as mentioned above, that stubble would be unable to reason effectively about cryptographic hashes or encryption because of properties like one-way functions and the avalanche effect. Likewise, checksum, hashing, and encryption aware fuzzing is another, orthogonal topic of interest [T].

In a traditional oracle attack, the adversary can derive a secret by testing one character at a time. This means an attacker can expect success within $\frac{mn}{2}$ tries on average, where m is the number of character possiblities and n is the length of the secret. Dumb fuzzers, which are unable to reason about the execution paths in a binary, do much worse by mutating random bytes and hoping to hit upon the secret by chance. This strategy can expect success within $\frac{m^n}{2}$ tries on average, since m-many possibilities must be tested for each byte in the secret independently. Stubble performs similarly to an attacker with access to an oracle, although because execution slices are backsolved its time complexity has a smaller constant than $\frac{m}{2}$. The expected number of tries is linear with secret length, as with an oracle attack, but stubble requires roughly two tries per byte, instead of having to test for all possible m.

Secrets of length 1 to 50 were tested at regular intervals. It appears that, aside from some constant overhead, that test cases generated and time required to crack the binary rise linearly with password length as expected. The constant overhead is probably introduced from overtaint, which itself likely comes from dynamically linked libraries which are opened and read, and thereby mistakenly introduced as tainted input.

Figure 3.1: Secret Guessing Test Cases

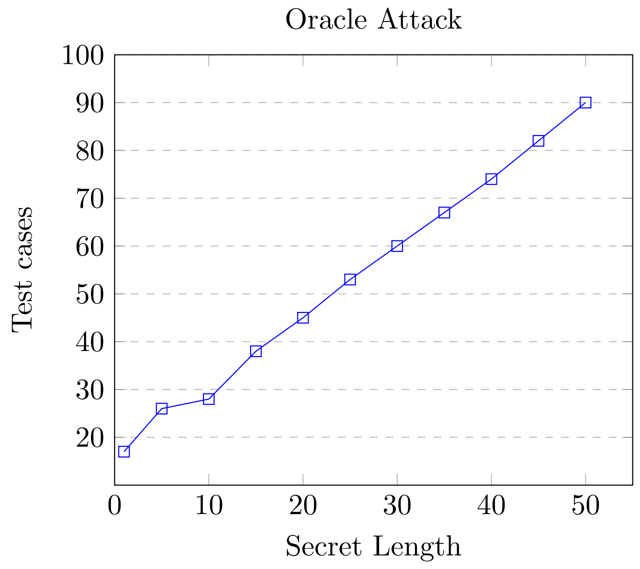
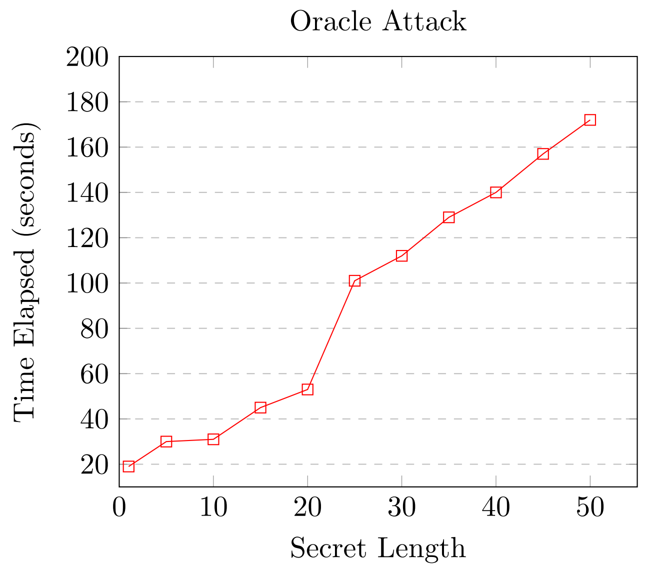


Figure 3.2: Secret Guessing Time Elapsed



CHAPTER 4

CONCLUSIONS

4.1 Applications

Some of the work involved in creating and testing Stubble feels spiritually similar to Netzob, a protocol reverse engineering tool which actively and passively infers protcol grammars and state machines [AQ]. Dynamic slice backsolving could be used to reverse engineer an unknown protcol by driving execution down increasingly newer and deeper paths. In order to do this, we must, by definition, more closely approximate valid input with each run, eventually mutating from random bytes to a valid protocol input.

TODO ways this technique could be used.

4.2 Future Work

Significant care has already been invested to keep Stubble easy to understand and maintain. Continuing to expand Stubble's functionality and developing it into a mature fuzzer is the long term goal. In order for Stubble to be useful to analysts, it must support the Windows operating system, more system calls and C library routines, and more vectors for user input. Intel Pin supports Windows, so in theory Stubble is compatible as well. In practice, Stubble currently makes assumptions about compilation and operating systems internals which are appropriate for Linux. In future development, these would be moved to a Linux branch, and a Windows branch would be forked, with common code re-used between the two. Stubble also only instruments the read syscall. Its features would need to be greatly expanded to handle both more relevant Linux system calls and their Windows equivalents. In addition, some system calls not only return success statuses, but also have side effects, which must be accounted for [C]. Lastly, the only input vector captured by Stubble currently is a read on a user supplied input file. There are many other ways taint could be introduced into a program, such as command line arguments, stdin, and input devices like keyboards and mice. Accurately recording alternative sources of user input has proved difficult. For example, command line arguments in Linux are pushed onto the stack before the main routine is called. If we instrument the main routine, we avoid catching all sorts of noisy reads on files that are unlikely to introduce taint, like system libraries. However, this makes argv inaccessible. Instrumenting the entire binary, dynamically loaded libraries and all, grabs argv, but introduces lots of noise. Better semantic for differentiating between noise and signal will be a necessity, for instance by maintaining a hardcoded list of names and sizes of standard libraries, so that they can be ignored.

TODO better "vuln disc." conditions like segfaults, taint reaching EIP.

CHAPTER 5

REFERENCES

A Bekrar, Sofia, et al. "A taint based approach for smart fuzzing." Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE, 2012.

B DeMott, J., Enbody, R., and Punch, W. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing", BlackHat and Defcon 2007.

C Schwartz, E.J., Avgerinos, T., Brumley, D. "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)." 2010 IEEE Symposium on Security and Privacy.

D Clause, J. Li, W., Orso, A. "Dytan: a generic dynamic taint analysis framework". 2007 Int'l symposium on Software testing and analysis. ACM, 2007.

E Ajit, Beng Heng Ng Earlence Fernandes, Aluri Atul Prakash, and David Rodriguez Velazquez Zijiang Yang. "Beyond Instruction Level Taint Propagation." (2013).

F Yamaguchi, Fabian, Felix Lindner, and Konrad Rieck. "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning." Proceedings of the 5th USENIX conference on Offensive technologies. USENIX Association, 2011.

G Bao, Tao, et al. "Strict control dependence and its effect on dynamic information flow analyses." Proceedings of the 19th international symposium on Software testing and analysis. ACM, 2010.

H Cha, Sang Kil, et al. "Unleashing mayhem on binary code." Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012.

I Avgerinos, Thanassis, et al. "AEG: Automatic Exploit Generation." NDSS. Vol. 11. 2011.

J Chipounov, Vitaly, Volodymyr Kuznetsov, and George Candea. "S2E: A platform for in-vivo multi-path analysis of software systems." ACM SIGARCH Computer Architecture News 39.1 (2011): 265-278.

K Bellard, Fabrice. "QEMU, a Fast and Portable Dynamic Translator." USENIX Annual Technical Conference, FREENIX Track. 2005.

L Guillon, Christophe. "Program Instrumentation with QEMU." 1st International QEMU Users Forum. 2011.

M Luk, Chi-Keung, et al. "Pin: building customized program analysis tools with dynamic instrumentation." Acm Sigplan Notices. Vol. 40. No. 6. ACM, 2005.

N Salwan, Jonathan. Shell-storm.org http://shell-storm.org/blog/Taint-analysis-and-pattern-matching-with-Pin/

O King, James C. "Symbolic execution and program testing." Communications of the ACM 19.7 (1976): 385-394.

P De Moura, Leonardo, and Nikolaj Bjrner. "Z3: An efficient SMT solver." Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 2008. 337-340.

Q Godefroid, Patrice, Michael Y. Levin, and David Molnar. "Sage: Whitebox fuzzing for security testing." Queue 10.1 (2012): 20.

R Ma, Kin-Keung, et al. "Directed symbolic execution." Static Analysis. Springer Berlin Heidelberg, 2011. 95-111.

S Boyer, Robert S., Bernard Elspas, and Karl N. Levitt. "SELECTa formal system for testing and debugging programs by symbolic execution." ACM SigPlan Notices. Vol. 10. No. 6. ACM, 1975.

T Wang, Tielei, et al. "Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution." ACM Transactions on Information and System Security (TISSEC) 14.2 (2011): 15.

U Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." Communications of the ACM 56.2 (2013): 82-90.

V Cadar, Cristian, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." OSDI. Vol. 8. 2008.

W McCamant, Stephen, et al. Transformation-aware symbolic execution for system test generation. Tech. Rep. UCB/EECS-2013-125, University of California, Berkeley (Jun 2013), 2013.

X Majumdar, Rupak, and Koushik Sen. "Hybrid concolic testing." Software Engineering, 2007. ICSE 2007. 29th International Conference on. IEEE, 2007.

Y Sen, Koushik, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. Vol. 30. No. 5. ACM, 2005.

Z Eddington, Michael. "Peach fuzzing platform." Peach Fuzzer (2011).

AA Zhang, Xiangyu, Rajiv Gupta, and Youtao Zhang. "Precise dynamic slicing algorithms." Software Engineering, 2003. Proceedings. 25th International Conference on. IEEE, 2003.

AB Wang, Tielei, et al. "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection." Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010.

AC Sen, Koushik. "Concolic testing." Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering. ACM, 2007.

AD Saxena, Prateek, et al. "A symbolic execution framework for javascript." Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010.

AE Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_O

AF The WebKit Open Source Project. http://www.webkit.org/

AG Portokalidis, Georgios, Asia Slowinska, and Herbert Bos. "Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation." ACM SIGOPS Operating Systems Review. Vol. 40. No. 4. ACM, 2006.

AH Rescue, Data. "IDA Pro Disassembler." 2006-10-20). http://www. datarescue. com/idabase.

AI Garg, Sanjam, et al. "Candidate indistinguishability obfuscation and functional encryption for all circuits." Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on. IEEE, 2013.

AJ Oehlert, Peter. "Violating assumptions with fuzzing." Security & Privacy, IEEE 3.2 (2005): 58-62.

AK Gilmore, John, and Sten Shebs. "GDB internalsa guide to the internals of the GNU debugger." Found in the doc directory of gdb distribution version 5.1 (2000).

AL Bruening, Derek, and Qin Zhao. "DynamoRIO."

AM http://googleonlinesecurity.blogspot.com/2011/08/fuzzing-at-scale.html

AN http://www.pwn2own.com/, http://www.pwn2own.com/2014/03/pwn2own-2014-lineup/

AO http://www.forbes.com/sites/andygreenberg/2012/03/23/shopping-for-zero-days-an-price-list-for-hackers-secret-software-exploits/

AP Braden, R., Borman, D., and C. Partridge, "Computing the Internet checksum", RFC 1071, September 1988.

AQ Bossert, Georges, Frdric Guihry, and Guillaume Hiet. "Netzob: un outil pour la rtro-conception de protocoles de communication." Actes du Symposium sur la scurit des technologies de l'information et des communications. 2012.

BIOGRAPHICAL SKETCH

Clark Wood is a Scholarship for Service recipient studying computer science at Florida State University. His research interests include reverse engineering, cryptography, networking and their intersection.