Checksum-Aware Fuzzing Combined with Dynamic Taint Analysis and Symbolic Execution

TIELEI WANG and TAO WEI, Peking University
GUOFEI GU, Texas A&M University
WEI ZOU, Peking University

Fuzz testing has proven successful in finding security vulnerabilities in large programs. However, traditional fuzz testing tools have a well-known common drawback: they are ineffective if most generated inputs are rejected at the early stage of program running, especially when target programs employ checksum mechanisms to verify the integrity of inputs. This article presents TaintScope, an automatic fuzzing system using dynamic taint analysis and symbolic execution techniques, to tackle the above problem. TaintScope has several novel features: (1) TaintScope is a checksum-aware fuzzing tool. It can identify checksum fields in inputs, accurately locate checksum-based integrity checks by using branch profiling techniques, and bypass such checks via control flow alteration. Furthermore, it can fix checksum values in generated inputs using combined concrete and symbolic execution techniques. (2) TaintScope is a taint-based fuzzing tool working at the x86 binary level. Based on fine-grained dynamic taint tracing, TaintScope identifies the "hot bytes" in a well-formed input that are used in security-sensitive operations (e.g., invoking system/library calls), and then focuses on modifying such bytes with random or boundary values. (3) TaintScope is also a symbolic-execution-based fuzzing tool. It can symbolically evaluate a trace, reason about all possible values that can execute the trace, and then detect potential vulnerabilities on the trace.

We evaluate TaintScope on a number of large real-world applications. Experimental results show that TaintScope can accurately locate the checksum checks in programs and dramatically improve the effectiveness of fuzz testing. TaintScope has already found 30 previously unknown vulnerabilities in several widely used applications, including Adobe Acrobat, Flash Player, Google Picasa, and Microsoft Paint. Most of these severe vulnerabilities have been confirmed by Secunia and oCERT, and assigned CVE identifiers (such as CVE-2009-1882, CVE-2009-2688). Vendor patches have been released or are in preparation based on our reports.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools and symbolic execution*

General Terms: Security, Languages

Additional Key Words and Phrases: Vulnerability detection, checksum-aware fuzzing, taint analysis, symbolic execution

ACM Reference Format:
Wang, T., Wei, T., Gu, G., and Zou, W. 2011. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. ACM Trans. Inf. Syst. Secur. 14, 2, Article 15 (September 2011), 28 pages.
DOI = 10.1145/2019599.2019600 http://doi.acm.org/10.1145/2019599.2019600

This work is supported by the National Natural Science Foundation of China under the grant No. 61003216 and the project "Mobile Internet Mal-behavior Detection Platform based on Cloud Computing (2010)" granted by the National Development and Reform Commission InfoSec Foundation.
Author's address: T. Wang, T. Wei, and W. Zou (corresponding author), Laboratory of Internet Security Technology and Institute of Computer Science and Technology, Peking University, Beijing, China, 100871; email: zou_wei@pku.edu.cn; G. Gu, Department of Computer Science and Engineering, Texas A&M University.

1. INTRODUCTION

As a well-known software testing technique, fuzz testing or fuzzing [Miller et al. 1990] has proven successful in finding bugs and security vulnerabilities in large software. The idea behind fuzzing is very simple: generating malformed inputs and feeding them to an application; if the application crashes or hangs, a potential bug/vulnerability is detected.

A number of severe software vulnerabilities have been revealed by fuzzing techniques [Miller et al. 1990; Oehlert 2005; Sutton et al. 2007]. For example, with the help of fuzzing tools, "Month of Browser Bugs" (http://browserfun.blogspot.com) and "Month of Kernel Bugs" (http://projects.info-pull.com/mokb) published bugs in various browsers and kernels on a daily basis for the month of July and November in 2006.

Since exhaustive enumeration of an application's input space is typically infeasible, there are two main approaches to obtain malformed inputs: *data mutation* and *data generation* [Oehlert 2005]. Mutation-based fuzzing tools generate test cases by randomly modifying well-formed inputs. However, most inputs from such blind modifications will be dropped at an early stage of program running if the target program employs a checksum mechanism to verify the integrity of inputs. The effectiveness of these fuzzing tools is heavily limited by checksum-based integrity checks.

Recently, symbolic execution and constraint-solving-based whitebox fuzzing systems and unit testing tools (such as KLEE [Cadar et al. 2008], SAGE [Godefroid et al. 2008a], SmartFuzz [Molnar et al. 2009], EXE [Cadar et al. 2008], CUTE [Sen et al. 2005], DART [Godefroid et al. 2005]) can treat all program inputs with symbolic values, gather input constraints on a program trace and generate new inputs that can drive program executions along different traces. These systems are able to provide good code coverage and have proven to highly improve the effectiveness of traditional fuzzing tools. However, since many complex checksum algorithms such as cryptographic hash functions are not invertible [Stallings 2005], current symbolic execution engines and constraint solvers still cannot accurately generate and solve the constraints that describe the complete process of these checksum algorithms. In a word, such whitebox fuzzing systems cannot automatically generate inputs that satisfy the checksum-based integrity constraints.

For the fuzzing tools (such as SPIKE[1] and Peach[2]) that construct malformed input data from predefined format specifications, the cost of generating production rules is expensive, especially when the format specifications are undocumented and the source code of the application is not available. Recently, several protocol reverse engineering techniques [Caballero et al. 2007; Comparetti et al. 2009; Cui et al. 2008; Lin et al. 2008; Wondracek et al. 2008] are proposed to automatically extract input format specification (even protocol state machine) and translate them into fuzzing specifications. However, they are not able to reverse engineer the checksum algorithms or locate the checksum check points. Thus, the constructed input according to such reverse-engineered protocol specification still can be rejected by integrity checks. In addition, none of such systems explicitly discuss how to bypass checksum checks for fuzzing.

This article presents TaintScope, a checksum-aware fuzzing system based on dynamic taint analysis and symbolic execution. The key idea behind TaintScope is that the taint propagation information during program execution can be used to detect and

[1]SPIKE. Spike Fuzzing Platform. http://www.immunitysec.com/resourcesfreesoftware.shtml.
[2]Peach. Fuzzing Platform. http://peachfuzz.sourceforge.net/.

bypass checksum-based integrity checks, and to direct malformed test case generation. TaintScope can further fix checksum fields in malformed test cases by using dynamic symbolic execution. Specifically, TaintScope has the following features.

First, TaintScope is a *checksum-aware* fuzzing tool. TaintScope can locate program branches corresponding to checksum related tests. Then, TaintScope can enforce the alteration of the target program's execution at located checksum test points, even if the input test cases violate checksum checks. We call this *checksum-aware fuzzing*. Checksum-aware fuzzing can prevent generated test cases from being prematurely dropped, and this feature is helpful to trigger subtle errors in the rest of the program. For the crashed test cases, TaintScope can fix their checksum fields using dynamic symbolic execution. Instead of treating all input bytes as symbolic values, TaintScope only treats the checksum fields in test cases as symbolic values (i.e., leaves the majority input bytes as concrete values), and collects trace constraints on checksum fields. Original complex trace constraints are simplified to simple constraints. By solving such simple constraints, TaintScope can repair generated test cases.

Second, TaintScope implements *taint-based* fuzzing [Ganesh et al. 2009; Zalewski 2007]. Based on fine-grained taint analysis at the byte level, TaintScope is able to accurately identify which input bytes can flow into security-sensitive points (e.g., memory allocation function `malloc()`, string manipulation function `strcpy()`). We refer to such input bytes as hot bytes. Given a well-formed input, instead of blindly modifying the whole input, TaintScope focuses on modifying hot bytes with random values or boundary values.

Third, TaintScope also partially implements *whitebox* fuzzing [Godefroid et al. 2008a; Molnar et al. 2009], complementary to taint-based fuzzing. The main advantages of whitebox fuzzing are: (1) it can automatically generate inputs that explore different paths, and (2) it can reason all possible input values on a trace to identify vulnerabilities. In this article, TaintScope mainly exploits the second advantage; that is, given an execution trace, TaintScope treats hot bytes in an original input as symbolic values and then uses offline dynamic symbolic execution to discover vulnerabilities on the trace. We refer to it as symbolic-execution-based fuzzing. Since TaintScope reasons about all possible hot bytes, it can detect some subtle vulnerabilities.

We evaluate TaintScope on a number of real-world applications. Experimental results show that our method can accurately locate the checksum checks in programs. TaintScope has already found 30 previously unknown vulnerabilities in several widely used applications from, among others, Microsoft, Google and Adobe. We have reported our findings to the vendors. Most of these vulnerabilities have been confirmed by Secunia and oCERT, and assigned CVE identifiers (such as CVE-2009-1882, CVE-2009-2688). Vendor patches have been released or are in preparation based on our reports.

The main contributions of this article can be summarized as follows: (i) We propose checksum-aware fuzzing, a novel heuristic technique to overcome the checksum problem for fuzz testing; (ii) We implement taint-based and symbolic-execution-based fuzzing methods at binary program level and successfully combine them with checksum-aware fuzzing; (iii) TaintScope detects 30 previously unknown vulnerabilities in several widely used applications.

The rest of this article is organized as follows. Section 2 explains the problem we address and presents an overview of our system. Section 3 details our checksum-aware fuzzing approach and Section 4 presents how to generate test cases and detect vulnerabilities on an execution. Section 5 describes the implementation details and Section 6 reports the evaluation results. Section 7 and Section 8 discuss limitations in the current implementation and related work, respectively. Finally, Section 9 concludes the article.

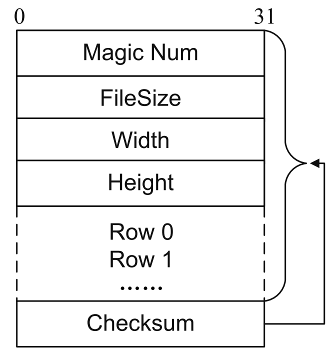

Fig. 1. Input format example.

2. OVERVIEW

2.1. Problem Scope and Terminologies

Checksums[3] are a common way to check the integrity of data and are widely used in network protocols (e.g., TCP/IP) and many file formats (e.g., ZLIB [Deutsch and Gailly 1996], PNG [Boutell 1997]). There are many sophisticated checksum algorithms used in practice, such as Adler-32, the CRC (cyclic redundancy checks) series, and MD5. In this article, we focus on checksums that are designed to protect against mainly *accidental* errors that may have been introduced during transmission or storage, instead of those designed to protect against *intentional* data alteration such as keyed cryptographic hash algorithms. We leave the latter as future work.

In general, the basic pattern to check the integrity of an input is to recompute a new checksum of the input and compare it with the checksum attached in the input. A mismatch indicates a corrupted input. For ease of representation, we use C_r and D to represent raw data in the checksum field and the rest of input data protected by the checksum field, respectively.

Without loss of generality, we assume that the checksum check condition is equivalent to the condition P: $Checksum(D) == T(C_r)$, where $Checksum()$ denotes the complete process of checksum algorithms and $T()$ denotes the transformation function that is used to transform C_r before integrity checks. For instance, the attached checksums are stored as octal numbers in the Tar format[4], or stored as hexadecimal numbers in the Intel HEX format[5]. These raw data C_r need to be converted into proper forms before being used in integrity checks. $T()$ is used to describe the transformation process.

We assume that there are special branch predicates in the program, corresponding to integrity checks P. The predicate P is always true/false with well-formed instances, whereas the predicate is always false/true with malformed instances. One of our goals is to accurately locate potential integrity check points in a binary program rather than identify the checksum algorithm themselves.

2.2. Motivating Example and System Overview

As a motivating example, Figure 1 shows an input format example. This input format presents a common image format: the `MagicNum` field declares the file format; the `FileSize` field indicates the real size of an input image; and the `Width` and `Height` fields represent the width and the height of an input image. In this format, image data are

[3]Checksum. From Wikipedia. http://en.wikipedia.org/wiki/Checksum.
[4]TAR. The gnu version of the tar archiving utility. http://www.gnu.org/software/tar/.
[5]Intel HEX. From Wikipedia. http://en.wikipedia.org/wiki/Intel_HEX.

```
 1  void decode_image(FILE* fd){
 2   ...
 3   int length           = get_length(fd);
 4   int recomputed_chksum = checksum(fd, length);
 5   int chksum_in_file    = get_checksum(fd);
 6   if(chksum_in_file != recomputed_chksum)
 7     error();
 8   int Width    = get_width(fd);
 9   int Height   = get_height(fd);
10   if(Width==0 || Height==0)
11     error();
12   int size = Width*Height*sizeof(int);
13   int* p   = malloc(size);
14   ...
15   for(i=0; i<Height;i++){// read ith row to p
16     read_row(p + Width*i, i, fd);
```

Fig. 2. Vulnerable code snippet.

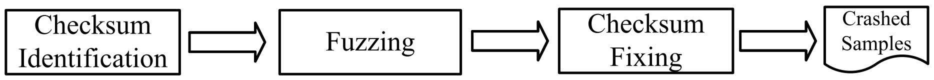

Fig. 3. TaintScope system overview.

stored in row-major order. Note that the format ends with a checksum field. A four-byte checksum is calculated on the preceding image content in the file for integrity protection.

Figure 2 shows an example code that parses inputs from the example format. The code first recomputes a checksum of an input file (line 4), reads the checksum stored in the file (line 5), and then compares the two values (line 6). If the two values mismatch, the code raises an error and exits immediately. Next, the code reads the `Width` and `Height` fields. After simple checks (line 10), the code allocates memory of size "`Width*Height*sizeof(int)`". Finally, the code reads image data row by row into the allocated buffer. However, a specially crafted image containing large width and height values can cause an integer overflow in the above expression (line 12) lead to an insufficient memory allocation at line 13. A heap overflow will eventually occur when the code reads image data into memory.

Traditional fuzzing methods such as randomly modifying a well-formed input file are unlikely to find such integer overflow vulnerability in Figure 2. Because a simple modification breaks the integrity of the original input, the failed integrity check at line 6 will cause most generated test cases to be rejected. Furthermore, even if there was no integrity check at line 6, traditional fuzzing methods would also be too inefficient. In this example, only eight bytes (the `Width` and `Height` fields) in an input file are key factors to trigger the integer overflow vulnerability; however, blind modifications need to explore the whole input space. The probability of exactly modifying the `Width` and `Height` fields is very low.

In contrast, TaintScope leverages taint propagation information during program execution to detect checksum checks and produce malformed inputs by taint-based fuzzing and symbolic-execution-based fuzzing. The high-level overview of TaintScope is shown in Figure 3. We use the program in Figure 2 to illustrate the basic workflow of TaintScope. Note that TaintScope neither depends on the source code nor on the input format specification.

Checksum Identification. TaintScope first runs the target program with well-formed and malformed inputs to locate the divergence points among these execution runs. The intuition is that checksum-based integrity checks behave like a classifier. While all well-formed inputs pass the integrity checks, most malformed inputs fail to pass because of integrity violations. We present the details of the checksum check point identification approach in Section 3.1.

Fuzzing. Once a checksum check point is located, TaintScope can modify the program by enforcing a control flow alteration at the check point to force the program to accept all inputs regardless of checksum check failures.

TaintScope implements taint-based fuzzing (see Section 4.1) to generate malformed inputs, and implements symbolic-execution-based fuzzing (see Section 4.2) to inspect execution traces and detect potential vulnerabilities on the traces.

In this example, taint-based fuzzing will focus on modifying the input bytes that flow into the function `malloc`. Since the checksum check has been disabled, the generated test cases are most likely to trigger the integer overflow.

The symbolic-execution-based fuzzing first records the execution trace of the target program under test on a well-formed input, then symbolically replays the trace and collects trace constraints on hot bytes. Symbolic-execution-based fuzzing uses a solver to check if collected trace constraints are insufficient to avoid vulnerabilities at security-sensitive operations. If the solver can produce a concrete solution to prove the constraints are insufficient, the symbolic-execution-based fuzzing creates a new input according to the solution to trigger the vulnerability.

In this example, collected trace constraints are `Width!=0 && Height!=0`, and the constraint on line 6 is ignored due to the control flow alteration. Before invoking the memory allocation function at line 13, since `size` is a symbolic value, TaintScope checks whether an integer overflow could occur in the calculation of `size`. Because the solver can find a solution (e.g., `Width=0x80000001`, `Height=0x4`), TaintScope directly generates a new input to trigger the integer overflow issue.

Checksum Fixing. For the test cases that crash the programs with control flow alteration, TaintScope needs to fix checksum fields in the test cases. Otherwise, the test cases cannot pass the checksum checks in the original program and trigger the vulnerabilities. Note that we do not apply checksum fixing in the previous phase for every fuzzing test case because the fixing is relatively time-consuming. It makes more sense to perform a delayed repair on only a small number of malformed test cases that actually cause the program to crash.

To pass the checksum check, the test case should satisfy the trace condition $Checksum(D) == T(C_r)$. If both the raw data in the checksum fields C_r and the rest of the data D are symbolic values, for complex checksum algorithms such as MD5, this constraint cannot be solved. However, TaintScope only treats the bytes in C_r as symbolic values. Thus, $Checksum(D)$ is a runtime determinable constant value. The complex constraint can be simplified to a simple one, which can be solved with current solvers such as Z3[6]. Finally, TaintScope fixes the checksum field according to the solver's solution. If the fixed test case can still cause the original program to crash, a potential vulnerability is detected.

3. CHECKSUM-AWARE FUZZING

Checksums are a simple error-detection scheme. The integrity of input data can be checked by recomputing a new checksum and comparing it with the checksum value stored in the input data. In general, checksum checks have three features.

First, since a checksum is usually used to protect a considerable number of contiguous bytes, the recomputed checksum value most likely depends on many input bytes. Meanwhile, the input checksum itself is usually a few bytes. Similar to Caballero

[6]Z3. Z3 SMT Solver. http://research.microsoft.com/en-us/um/redmond/project/z3/.

et al. [2010], we use *taint degree* to represent the number of input bytes that a value depends on. If the taint degree of a value is larger than a predefined threshold, we call it high-taint-degree; otherwise, we call it low-taint-degree. Furthermore, if the number of taint labels associated with the processor flags register (i.e., `eflags` register) exceeds a predefined threshold value, the conditional jump instruction that is affected by the flags register is called a high-taint-degree branch. Usually, a checksum check is a high-taint-degree branch and there are a high-taint-degree value (i.e., the recomputed checksum value) and a low-taint-degree value (i.e., the input checksum value) involved.

Second, checksum checks behave like a classifier: While all well-formed inputs pass the integrity checks, most malformed inputs fail to pass because of integrity violations. Thus, we assume that there are special branch predicates in the program, corresponding to integrity checks. When the program runs with well-formed inputs, these branch predicates are always True/False; however, when the program runs with malformed inputs, these branch predicates are always False/True.

Third, checksum computation often does not produce any side effects on the rest of execution if an input passes the checksum test [Caballero et al. 2010]. In other words, input checksum values are only used in checksum checks. Thus, for any input, checksum constraints can always be satisfied by choosing a right input checksum value.

Based on the observations above, our checksum-aware fuzzing approach consists of three key phases: locating checksum checks (Section 3.1), modifying and testing the program (Section 3.2), and repairing checksum fields (Section 3.3).

3.1. Locating Checksum Check Points and Checksum Fields

In this section, we describe how to locate checksum check points in a program and identify checksum fields in well-formed inputs.

3.1.1. Identifying High-Taint-Degree Branches. Given a well-formed input, Taint-Scope locates high-taint-degree branches based on fine-grained dynamic taint analysis. To this end all input data is tainted. TaintScope assigns each input byte a unique label. TaintScope identifies the source of taint data by intercepting relevant system calls. For example, for file tainting on Linux systems, TaintScope intercepts system calls such as `open`, `mmap`, `read`, `lseek`, and `close`. When the specified file is successfully opened, TaintScope records the returned file descriptor `fd`. Each time the system reads from file descriptor `fd`, TaintScope scans the input buffer, and assigns each byte in the input buffer with its offset in the file. According to the return value of `read` and `lseek`, TaintScope updates the file offset. A closed `fd` is not tracked any more.

TaintScope tracks the propagation of taint labels throughout the execution of the program according to data dependence relationships. All values written by an instruction are marked with the union of all taint labels associated with values used by the instruction. Note that the flags register is also considered. For example, assume that the taint labels associated with registers `eax` and `ebx` are {0x6, 0x7} and {0x8, 0x9}, respectively, after the execution of instruction "`ADD eax, ebx`", the taint labels on `eax` would be {0x6, 0x7, 0x8, 0x9}; since instruction `ADD` also affects the flags register, the taint labels on the flags register are updated to {0x6, 0x7, 0x8, 0x9}. For another example, assume the taint label associated with `ecx` is {0x100} and `0x8000000` is the base address of an untainted array, after the execution of this instruction "`MOV eax, [0x8000000+ecx*4]`", the `eax` would be associated with {0x100}. Currently, TaintScope does not consider control-flow dependencies.

Before the execution of each conditional jump instruction, TaintScope checks the taint labels associated with the flags register and determines whether the conditional jump instruction is a high-taint-degree branch. If so, TaintScope logs the conditional jump instruction and the input bytes it depends on.

3.1.2. Locating Divergence Points. Based on the high-taint-degree branch identification technique, TaintScope first runs the program with several well-formed inputs and identifies the always-taken and always-not-taken high-taint-degree branches. Specifically, TaintScope builds two sets $\mathcal{P}_1$ and $\mathcal{P}_0$, where $\mathcal{P}_1/\mathcal{P}_0$ includes the high-taint-degree conditional jump instructions that are always-taken/always-not-taken among all executions.

Next, TaintScope runs the program with some malformed inputs and also builds the always-taken and always-not-taken high-taint-degree branch sets $\mathcal{P}'_1$ and $\mathcal{P}'_0$. For each well-formed input, we randomly modify a byte each time that can affect conditional jump instructions in the set $\mathcal{P}_1 \cup \mathcal{P}_0$ to generate malformed inputs. TaintScope particularly tracks the propagation of the modified byte. Even if a conditional jump instruction is executed multiple times during a single run, TaintScope only takes into account the action of the conditional jump instruction when it is affected by the modified byte.

The reason that we need to specially track the modified bytes is to deal with input formats containing multiple checksum fields that are used to protect different parts of an input. For example, a PNG format image consists of many `chunks`, and each `chunk` has a CRC checksum. Even if we modify the bytes in the last `chunk`, other `chunks` can still pass the checksum checks. Therefore, the conditional jump instruction corresponding to the checksum check point has different actions in an execution. However, TaintScope only focuses on the action affected by the modified byte. This means that only the action of checksum test failure is noted.

Finally, TaintScope computes the set $(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)$. The conditional jump instructions in $(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)$ behave completely different when the target program runs with well-formed inputs and malformed inputs. TaintScope outputs such instructions as checksum check points.

A special case is when the checksum value is longer than 4 bytes (in 32-bit architectures). In this case, the program will most likely use multiple checks that are corresponding to multiple branches to compare the checksum in the input with the recomputed one. For example, an MD5 checksum is 16 byte long and the program may use four 32-bit comparisons or a `memcmp`-like operation to perform the check.

Intuitively, the multiple branches are most likely corresponding to compound Boolean conditions (i.e., conditions using logical operators such as `and` and `or`). The control flow graphs formed by such branches have special features and can be identified by means of 2-way branches structuring algorithms [Cifuentes 1994; Wei et al. 2007]. Based on control flow analysis, we can structure the corresponding branches to a compound conditional [Wei et al. 2007]. Once a branch in a compound conditional is considered as a checksum check, we will further check other branches in the compound conditional. If these branches perform checks on adjacent high-taint-degree memory locations, we take the whole compound conditional as a checksum test.

Moreover, a `memcmp`-like operation may translate into a string comparison instruction preceded by a repeat string operation prefix (e.g., `rep cmps`). In this case, our tool can accurately locate the checksum test point. However, a `memcmp`-like operation more likely translates into a complicated loop. TaintScope currently does not deal with this case, but in the future we can explore the side-condition identification method proposed by Caballero et al. [2010].

3.1.3. Identifying Checksum Fields. Based on the observation that the recomputed checksum value is usually a high-taint-degree value and the input checksum value is a low-taint-degree value, we use a method similar to Tupni [Cui et al. 2008] to identify checksum fields in well-formed inputs.

Specifically, for the conditional jump instructions in the set $(\mathcal{P}_1 \cap \mathcal{P}'_0) \cup (\mathcal{P}_0 \cap \mathcal{P}'_1)$, our tool backwards inspects instructions that can be used to perform a comparison (such as `test`, `sub` and `cmp`) and directly or indirectly affect these conditional jump instructions. Once an instruction uses a high-taint-degree value and a low-taint-degree value, the input bytes that the low-taint-degree value depends on are considered a checksum field.

Furthermore, since input checksums are not used any more once an input passes the test, our tool will rerun the program with well-formed inputs and specially track the identified checksum fields. Only when the identified checksum fields do not participate in any constraints after checksum check points, we ensure that we can enforce a control flow alteration at the identified checksum check points.

3.2. Binary Program Modification and Fuzzing

In this phase, TaintScope modifies the original program by enforcing a control flow alteration at the checksum check point. The goal of the modification is to disable the checksum test functionality in the program.

TaintScope implements an ELF/PE file format parser and directly locates and modifies the checksum check points in the raw binary file. If the conditional jump instruction is always-taken for well-formed inputs (e.g., “`jcc NormalExecution`”), TaintScope uses a direct jump instruction (e.g., “`jmp NormalExecution`”) to replace the conditional jump instruction; otherwise, if the conditional jump instruction is always-not-taken for well-formed inputs (e.g., “`jcc checksumErr`”), TaintScope uses `nop` instructions to override the conditional jump instruction. The modified program will “accept” all kinds of input regardless of checksum test failures.

Furthermore, TaintScope will run the modified program with the malformed inputs and monitor for crashes. We discuss test case generation and vulnerability detection methods in Section 4.

3.3. Repairing Test Cases Using Combined Concrete and Symbolic Execution

For the malformed inputs that can crash the modified program, TaintScope needs to fix checksum fields in them so that they can pass the checksum check in the original program.

TaintScope implements offline dynamic symbolic execution [Godefroid et al. 2008a]; that is, it records the execution trace and then symbolically evaluates the execution trace to generate a correct checksum value for the malformed input.

Tracing. TaintScope first runs the original program with the malformed input, and records the execution trace. For each executed instruction, the recorded information includes the addresses and values of memory locations the instruction accesses, the values of modified registers, and the instruction’s address.

The effects of some interesting system calls are also logged. For example, for the file read system calls that read data from the input file into a memory buffer, the recorded information includes the file position, the starting address of the buffer, and the number of bytes read. This information will be used to build initial symbolic values during the replay phase.

Replay. Next, TaintScope symbolically evaluates the execution trace, gathers trace constraints on checksum fields, and generates a new input with a correct checksum value.

TaintScope maintains a symbolic memory environment, which records a map $\mathcal{M}$ from concrete addresses/registers to symbolic expressions, and a set of symbolic path conditions. Initially, there are no symbolic values in the symbolic memory environment. When the input data in the checksum field are read into memory, TaintScope creates a symbolic value (i.e., an 8-bit vector) for each input byte, and inserts the mappings from the concrete addresses to bit vectors into the map $\mathcal{M}$.

For an `N` bytes memory read operation from a concrete address `a`, TaintScope looks up the concrete addresses `a`, `a+1`, ... and `a+N-1` in $\mathcal{M}$ and returns a concatenated bit-vector $\mathcal{M}$`[a+N-1]@...@`$\mathcal{M}$`[a]`. `N` bytes memory writes to a concrete address are handled similarly.

Although other dynamic symbolic execution tools such as Cadar et al. [2008] and Elkarablieh et al. [2009] can reason about symbolic pointers, memory-object-specific information such as a memory object's starting address and size is required. Without source code and debug information, we can collect heap objects information at runtime by monitoring memory allocation functions. However, recovering information about local variables (i.e., local structures and arrays) and global preallocated variables at the x86 binary level is very challenging [Balakrishnan and Reps 2010], because the binary code does not contain bounds for individual variables. BitScope [Brumley et al. 2007] can deal with symbolic addresses at the x86 binary level, but it needs to use a solver to determine the range of possible values of each symbolic address.

Currently, TaintScope only handles global array reads that contain a symbolic index. TaintScope uses a simple approach to infer the sizes of global arrays. TaintScope first uses the IDA Pro[7] commercial disassembler to analyze the program, and implements an IDA Pro plug-in to export all global variables identified by IDA Pro. The size of a global variable is the space between current variable and the following variable. TaintScope builds bit vector arrays for identified global arrays. For a memory read from a symbolic address, if the concrete address of the symbolic address points to a global array, TaintScope creates an array read expression. We leave full symbolic pointer reasoning as future work.

Since TaintScope only treats the checksum fields as symbolic values and leaves the majority input bytes as concrete values, the collected trace constraints are essentially equal to a simple one $c == T(C_r)$, where c is a runtime determinable constant value. Solving the constraints will generate a correct checksum field for the malformed input. We can re-run the program with the fixed input. If the program still crashes, a potential vulnerability is detected.

4. TEST CASE GENERATION AND VULNERABILITY DETECTION

To more effectively test the modified program and identify vulnerabilities on execution traces, TaintScope implements taint-based fuzzing and symbolic-execution-based fuzzing techniques.

4.1. Taint-Based Fuzzing

Unlike traditional fuzzing tools that blindly change parts of a well-formed input, taint-based fuzzing [Ganesh et al. 2009] specially focuses on modifying the input bytes that affect the values used in security-sensitive operations.

Based on the fine-grained dynamic taint analysis, TaintScope can identify hot bytes in a well-formed input. TaintScope considers input bytes that can pollute the arguments of memory allocation functions (e.g., `malloc`, `realloc`) and string functions (e.g.,

[7]IdaPRO. www.hex-rays.com/idapro/idadown.htm.

```
1. int x, y;
2. x = read_16bits();
3. y = read_16bits();
4. if(x>0 && y>0 && x*y*4>x && x*y*4>y*4){
5.   p = malloc(x*y*4);
6.   if(p==NULL) return 0;
7. ......
```

Fig. 4. Pseudocode for the vulnerability CVE-2010-2170.

strcpy, strcat) to be hot bytes. Analysts can also configure TaintScope to check other functions. Hot bytes that can influence memory allocation functions are set to small, large or negative integer values; hot bytes that flow into string functions are partially replaced by special strings (e.g., format directives %n, %p) or a long random string. Format directives are closely related to format string vulnerabilities and long strings may cause buffer overflow issues.

Taint-based fuzzing can dramatically reduce the size of the mutation space because usually only a small portion of input data is hot bytes. Due to the similarity to well-formed inputs, generated test cases satisfy many primitive constraints and have a high probability to test code within the entire program. In addition, since extreme values in generated test cases may directly affect the arguments of system calls or other important APIs, such malformed test cases have a high probability to trigger potential vulnerabilities.

On the other hand, taint-based fuzzing has an important limitation. Since taint-based fuzzing does not consider trace conditions, it cannot guarantee that new inputs can still reach the security-sensitive operations.

4.2. Symbolic-Execution-Based Fuzzing

TaintScope partially implements whitebox fuzzing [Godefroid et al. 2008a; Molnar et al. 2009]. In general, whitebox fuzzing can create new inputs to explore different paths (i.e., test case generation) and detect vulnerabilities on a single trace (i.e., vulnerability discovery). TaintScope mainly exploits the vulnerability discovery feature, which is referred as symbolic-execution-based fuzzing in this article. Once a vulnerability is detected, symbolic-execution-based fuzzing can directly construct a test case to trigger the vulnerability.

As a complementary technique to taint-based fuzzing, we use the code snippet in Figure 4 to show the advantage of symbolic-execution-based fuzzing. The code performs complicated checks on variables x and y at line 4. Although well-formed inputs can pass the validation, a subtle integer overflow exists when calculating x*y*4. In other words, even if a path is executed multiple times, some subtle vulnerabilities may be still hidden. It is very hard for taint-based fuzzing to produce such corner test cases that can pass the validation checks and trigger the vulnerability at the same time.

Given an execution trace, TaintScope can reason about all possible values that the path could be run with and check if trace constraints properly validate inputs. Specifically, TaintScope extends offline dynamic symbolic execution technique used in checksum repairing to vulnerability detection. TaintScope treats hot bytes as symbolic values and collects trace constraints on hot bytes. When meeting security-sensitive operations, TaintScope creates a predicate to describe the security property of the operations, and checks whether the trace constraints and the predicate are satisfiable. If the solver can find a concrete solution to prove that some input values can pass the constraints and violate a security property, TaintScope will create a new input according to the solution. If the new input can cause the program to crash, TaintScope finds a vulnerability.

Specifically, TaintScope can detect integer overflow and buffer overflow issues. To detect integer overflow vulnerabilities, TaintScope employs a similar method used in IntScope [Wang et al. 2009a]. If the size argument of memory allocation functions is a symbolic arithmetic expression, TaintScope emits a predicate that forces the arithmetic expression to overflow, and then uses the solver to check whether the predicate is satisfiable under a given set of collected trace constraints. A solution means an input likely to cause an integer overflow. SmartFuzz [Molnar et al. 2009] can also detect integer bugs at the binary program level based on type inference. While SmartFuzz checks whether each arithmetic expression could overflow, TaintScope only checks the arithmetic expressions used in security-sensitive operations.

In practice, we find that memory allocation functions will return NULL if the size argument is too large. A NULL pointer check can usually detect memory allocation errors and avoid the program crash. If the program does not crash, the integer overflow vulnerability cannot be monitored. Thus, TaintScope checks whether the symbolic arithmetic expression could overflow and the calculation result could be less than a threshold value at the same time. The threshold value is chosen empirically.

To detect buffer overflow issues, TaintScope employs the method similar to Elkarablieh et al. [2009]. In general, for heap objects, TaintScope monitors memory allocation functions to obtain their sizes and starting addresses information; TaintScope uses the sizes of stack frames to approximate the bounds of local objects. Similar to Elkarablieh et al. [2009], TaintScope checks whether a symbolic address is confined in a proper range.

5. SYSTEM IMPLEMENTATION

We have implemented a prototype of TaintScope, which can run on both Windows and Linux systems. We introduce the implementation details in this section.

Taint Analysis and Tracing. TaintScope implements fine-grained taint analysis on top of the PIN [Luk et al. 2005] binary instrumentation platform. TaintScope intercepts low-level system calls and identifies the system calls relevant to input data.

TaintScope makes use of instruction instrumentation to track the propagation of taint data. By routine instrumentation, TaintScope is able to check the context of specified APIs before the execution of such APIs.

TaintScope can record an execution trace. To reduce the size of the trace file, all executed instructions' native codes are logged only once and stored in a separate file. For example, assume the instruction `xor ebx, ebx` (native code `0x33 0xFF`) is executed multiple times, the native code is logged only once. Hence, the size of the trace file is roughly proportional to the number of recorded instructions. Also, TaintScope can only log the trace after input data are read into memory or ignore the instructions in some GUI libraries that do not affect the propagation of the input data. To support multithreaded applications, TaintScope builds separate trace files for each thread.

Dynamic Symbolic Execution and Constraint Solving. Because it is a significant challenge to directly evaluate x86 instructions due to the complexity of the x86 instruction set [Molnar et al. 2009; Brumley et al. 2007], TaintScope uses the VEX library in Valgrind package [Nethercote and Seward 2007] to convert each x86 instruction in the trace file into VEX intermediate representation (IR) and then performs dynamic symbolic execution on the VEX IR. We have ported the VEX library to the Windows platform.[8]

[8]We can provide the VEX library running on Windows systems on demand.

A single x86 instruction will be converted into a VEX block that contains a list of VEX statements (i.e., `IRStmt`). A VEX statement is an operation with side-effects, such as register writes, memory stores, and assignments to temporaries. VEX statements contain VEX expressions (i.e., `IRExpr`), which represent constants, register reads, memory loads, or arithmetic operations. We refer readers to [Nethercote and Seward 2007] for more detailed information about VEX IR.

The shortcoming of VEX IR is that a single x86 instruction may be converted into multiple IR statements. For instance, the instruction "`adc esi, edx`" will expand to 16 VEX statements. In addition, a few infrequently used x86 instructions cannot be transformed by VEX library, such as "`lsl`, `STMXCSR`". We just ignore these instructions.

We use Z3[9] as our solver. When input data are read into memory, TaintScope uses `Z3_mk_const` to create bit-vectors variables (i.e., symbolic values). Z3 provides rich APIs to build complex bit-vector arithmetic operations and also supports sign-sensitive relational operations (signed/unsigned comparisons).

6. EVALUATION

In this section, we present our evaluation results. In Section 6.1 we present the experiment setup. In Section 6.2 we evaluate the effectiveness of our checksum check point identification approach. In Section 6.3, we evaluate the accuracy of checksum field identification and the capability of repairing checksum fields for given test cases. In Section 6.4, we evaluate the efficiency of our taint analysis and measure the portion of hot bytes in well-formed inputs. Section 6.5 gives a case study of our symbolic-execution-based fuzzing and shows how to apply our symbolic-execution-based fuzzing to Adobe Flash Player. In Section 6.6, we show the vulnerabilities we detected in several widely used applications.

6.1. Experiment Setup

We apply TaintScope to a large number of real-world applications, which are summarized in Table I. The "OS" column in Table I indicates the operating system that the applications run on.

For applications running on the Windows platform, our experiments are conducted on a machine with Intel Core 2 Duo CPU at 3.0 GHz and 3.25GB memory, running Windows XP Professional SP3; for applications on the Linux platform, our experiments are conducted on a machine with Intel Core 2 Duo CPU at 2.4 GHz and 4.0GB memory, running Fedora Core 10.

6.2. Checksum Check Points Identification

In this section, we evaluate the effectiveness of our checksum check point identification method.

File formats and checksum algorithms. We choose six known file formats,[10] as shown in the "Format" column in Table II. These six file formats employ different checksum algorithms to calculate checksum values.

— *PNG*, a popular image format with lossless compression, supports two main types of integrity-checking. First, PNG images are divided into logical data `chunks`, and each `chunk` has an associated CRC stored with it. The integrity of an image can be easily tested without decoding the image. Second, compressed data streams within

[9]See footnote 6.
[10]Note that TaintScope does not depend on knowing either file formats or checksum algorithms.

Table I. A List of Applications Used in Our Experiment

Category	Application	Version	OS
Image Viewer	Google Picasa	3.1.0	Windows
	Adobe Acrobat	9.1.3	Windows
	ImageMagick	6.5.2-7	Linux
	Microsoft Paint	5.1	Windows
Media Player	MPlayer	SVN-28979	Linux
	Gstreamer	0.10.15	Linux
	Adobe Flash Player	10.0.45	Windows
	Winamp	5.552	Windows
Web Browser	Amaya	11.1	Windows
	Dillo	2.1.1	Linux
Network Tool	Snort	2.8.4.1	Linux
	Tcpdump	4.0.0	Linux
Other	libtiff	3.8.2	Linux
	XEmacs	21.4.22	Linux
	wxWidgets	2.8.10	Linux
	ClamAV	0.95.2	Linux
	GNU Tar	1.22	Linux
	objcopy	2.17	Linux
	Open-vcdiff	0.6	Linux

PNG are stored in the zlib format [Deutsch and Gailly 1996]. Zlib format stores an Adler32 checksum value of uncompressed data.

— *PCAP*[11], a widely used format for dumping network packet traces, is supported by many packet analyzers, such as Tcpdump, Snort, and Wireshark. Although a PCAP file itself does not contain checksum fields, when parsing a PCAP file, packet analyzers need to check the checksums in TCP/UDP packets in the PCAP file.

— *CVD*[12] is an acronym for ClamAV Virus Database. A CVD file has a 512-bytes header structure, which stores an MD5 checksum value of the whole CVD file. When loading a CVD file, ClamAV first checks the integrity of the CVD file.

— *VCDIFF* (Generic Differencing and Compression Data Format) [Korn et al. 2002] is a general, efficient and portable data format suitable for delta encoding. Given a "source file" and a "target file," the VCDIFF format is used to describe the delta file. During computing the delta file, the input files are partitioned into chunks. `Open-vcdiff` project[13] extends the standard VCDIFF format and attaches an Adler32 checksum of the delta data for each chunk. Since the **VCDIFF** format encodes unsigned integer values using a variable-sized format, Adler32 checksums do not have a fixed length in a delta file.

— *Tar*[14] archive format is widely used in Unixlike systems. A tar archive file is the concatenation of one or more files. Each file in a "tar" package is preceded by a 512-byte header that contains a checksum value for the whole header. The checksum value is calculated by taking the sum of the unsigned byte values of the header block and stored as octal numbers followed by a NULL character and a space character.

[11]PCAP. Next Generation Dump File Format. www.winpcap.org/ntar/draft/PCAP-DumpFileFormat.html.
[12]CVD. ClamAV Virus Database. http://clamav.net/doc/latest/html/node54.html.
[13]Open-VCDiff. An encoder/decoder for the VCDIFF format. http://code.google.com/p/open-vcdiff/.
[14]See footnote 4.

Table II. High-Taint-Degree Branches Identification Results

Executable	Format	Checksum	|W|	HTDB8	HTDB16	HTDB24	HTDB32
PicasaPhotoViewer	PNG	CRC	3	1,079	881	544	544
Acrobat			3	16,472	10,537	10,537	10,537
Snort	PCAP	TCP/IP	3	2	2	1	1
Tcpdump			3	8	2	1	1
sigtool	CVD	MD5	1	39	39	32	2
vcdiff	VCDIFF	Adler32	3	19	1	1	0
GNU Tar	Tar Archive	Tar	3	34	34	28	5
objcopy	Intel HEX	Intel HEX	3	38	23	23	23

We report the number of well-formed inputs we used (|W|) and the number of high-taint-degree branches (HTDB) with threshold values of 8, 16, 24, and 32.

— *Intel HEX*[15] is a text format, with each line containing hexadecimal values encoding a sequence of data and a checksum value of the data. The checksum value is calculated by adding together the hex-encoded bytes, then leaving only the least significant byte of the result, and finally making a two's complement.

Evaluation Methodology. We test eight applications, as shown in the first column in Table II. These eight applications can process file formats mentioned above. Specifically,

— We input PNG images to two "closed source" image manipulation applications, `Picasa` and `Acrobat`, and apply TaintScope to locate potential CRC check points in the two applications.
— We input PCAP files to `Tcpdump` and `Snort`. The well-formed PCAP files are obtained by capturing network traffic from our local machine. We specify "`-v tcp -r`" options, which enable `Tcpdump` and `Snort` to read TCP packets from a saved PCAP file and perform packet integrity checks such as verifying the IP header and TCP checksums.
— With the "`--info`" option, `sigtool` included in the `ClamAV` package can verify the integrity of a given CVD file and display detailed information about the file. The well-formed CVD file used in our experiment is `daily.cvd` in `ClamAV` package.
— For the VCDIFF format, we test the application `Open-vcdiff`. By specifying `-checksum` option, we first use `Open-vcdiff` to create three delta files. Then, we apply TaintScope to locate the checksum check point when `Open-vcdiff` decodes these delta files.
— For the Tar Archive format, we test GNU `Tar`. We first use GNU `Tar` to create three well-formed tar archives. Then, we apply TaintScope to locate checksum checks when GNU `Tar` extracts files from the tar archives.
— For the Intel HEX format, we test GNU `objcopy`, which can copy the contents of an object file to another. In particular, GNU `objcopy` can translate an object file into another object file in a different format. We apply TaintScope to locate checksum checks when GNU `objcopy` translates an Intel HEX object file into other formats.

Experimental Results. The first step of our method is to run the target programs with well-formed inputs and identify high-taint-degree branches. Table II summarizes the

[15]See footnote 5.

Table III. Divergence Point Identification Results

Executable	|M|	CCP8	CCP16	CCP24	CCP32
PicasaPhotoViewer	9	1	1	1	1
Acrobat	9	1	1	1	1
Snort	9	2	2	1*	1*
Tcpdump	9	2	2	1*	1*
sigtool	3	1	1	1	1
vcdiff	9	1	1	1	0*
GNU Tar	9	1	1	1	1
objcopy	9	1	1	1	1
Detected?		√	√	Miss 2	Miss 3

We report the number of malformed inputs (|M|) and the number of checksum check points (CCP) identified with threshold values of 8, 16, 24, and 32. A * sign means that a checksum check point is missed.

results. The fourth column (|W|) shows the number of well-formed inputs we use for each program. We measure the number of high-taint-degree branches (HTDB) with the threshold values of 8, 16, 24 and 32, as shown in the rightmost four columns in Table II.

In general, with the increasing of the threshold value, the number of high-taint-degree branches is decreasing. For the PNG format, TaintScope identifies a large number of high-taint-degree branches. The main reason is that image data in PNG format is compressed. Any byte in the decompressed data stream may depend on the entire compressed data. Any check on the decompressed data causes a high-taint-degree branch.

Next, TaintScope runs the target programs with malformed test cases and locates the divergence point among all executions. Table III summarizes the results. The second column (|M|) presents the number of malformed inputs we used. We measure the number of located checksum check points (CCP) with the threshold values of 8, 16, 24 and 32, as shown in the rightmost four columns in Table III.

Manual inspection reveals that TaintScope can accurately locate the checksum check points with the threshold values 8 and 16. For `Acrobat`, TaintScope accurately locates the CRC check point in a binary file `ImageConversion.api`. Due to the integrity check failure, `Acrobat` refuses to decompress image data in malformed PNG images and exits immediately. Thus, TaintScope does not locate the Adler32 checksum points.

However, TaintScope misses some checksum check points if we set the threshold value to 24 and 32. The * signs in Table III mean that TaintScope misses a checksum check point. For instance, `Tcpdump` and `Snort` first verify the IP header checksum when parsing an IP packet. Since the IP Header checksum is computed on the header fields [Postel 1981], the taint-degree of IP checksum check point is 20 bytes. Both the threshold values 24 and 32 are too large. Similarly, the threshold value 32 causes that TaintScope misses the checksum point in `vcdiff`. In our system, we have chosen 16 as the threshold value by default because it well balances the coverage and efficiency of our algorithm.

The time cost of this phase mainly depends on how many test cases we use and the time spent on each test case. According to our experience, several well-formed test cases and malformed test cases are enough to locate the checksum check points in target programs. In general, processing a test case usually needs a few minutes (see Table V). We will report the performance of our taint analysis in Section 6.4. Overall, this phase can typically be done in tens of minutes.

Table IV. Checksum Fields Identification and Fixing Results

Executable	File Format	# Checksums	Checksum Length	Repaired?	Time (s)
Acrobat	PNG	4	4	√	41
PicasaPhotoViewer					106
Tcpdump	PCAP	8	2	√	58
vcdiff	VCDIFF	1	5	√	9
tar	Tar Archive	3	8	√	226
objcopy	Intel HEX	4	2	√	121

6.3. Checksum Fields Identification and Repairing

We further evaluate the accuracy of checksum field identification and the capability of repairing checksum fields for malformed test cases. We test six applications and file formats. The results are shown in Table IV.

The third column in Table IV presents the number of checksum fields identified by TaintScope in an input; the fourth column means the size (in bytes) of each checksum field identified by TaintScope.

Manual inspection reveals that TaintScope accurately identifies the number and the size of checksum fields in each well-formed instance. For example, TaintScope identifies four 4-byte checksum fields in a PNG image. We use 010editor[16], a hex editor with binary templates, to parse the PNG image for verification. The output of 010editor indicates that there are four chunks (e.g., `IHDR`, `PLTE`, `IDAT`, and `IEND`) in the PNG image and each chunk has a 4-byte CRC checksum field. TaintScope identifies eight 2-byte checksum fields in the well-formed PCAP file. For verification, we use Wireshark[17] to parse the PCAP file: there are four TCP packets, and each TCP packet has an IP checksum field and a TCP checksum field.

The fifth column in Table IV indicates that TaintScope can automatically generate valid checksum fields for given test cases. The experiments proceed as follows. First, we deliberately alter the bytes in checksum fields in such well-formed inputs to generate malformed ones. Note that here we modify checksum fields instead of regular data fields. The reason is that we want to show TaintScope can generate correct/valid checksums for which we have ground truth (the original ones). Next, we input these malformed inputs to the corresponding applications, and use TaintScope to record and replay the execution traces. TaintScope treats the bytes in checksum fields as symbolic values. After symbolically re-executing the traces, TaintScope generates new test cases. In our experiments, these new generated test cases are identical with those original well-formed ones, that is, TaintScope generates correct checksum fields.

The sixth column in Table IV shows the time (in seconds) TaintScope takes to replay recorded traces and solve path constraints. In our experiments, the time spent on fixing a given test case ranges from tens of seconds to several minutes.

In short, TaintScope can accurately identify checksum fields and automatically generate valid checksum fields for given test cases.

6.4. Hot Bytes Identification

In this set of experiments, we evaluate the performance of the fine-grained taint analysis and measure the portion of hot bytes in well-formed inputs. Specially, our previous research [Wang et al. 2009a] found that many integer overflow vulnerabilities are

[16]010editor. The text/hex editor. http://www.sweetscape.com/010editor/.
[17]Wireshark. Network Protocol Analyzer. http://wireshark.org/.

Table V. Hot Bytes Identification Results

Executable	Input Format	Input Size (Bytes)	# Hot Bytes	# x86 Inst	Run Time
Display	TIFF	5,778	18	191,759,211	2m53s
		2,020	18	82,640,260	1m30s
	PNG	5,149	9	19,051,746	1m54s
		1,250	29	47,246,043	1m8s
	JPEG	6,617	11	48,983,897	1m13s
		6,934	9	48,823,905	1m11s
PicasaPhoto Viewer.exe	GIF	3,190	14	304,993,501	1m25s
		6,529	43	536,938,567	2m57s
	PNG	2,730	18	712,021,776	5m16s
		1,362	16	660,183,239	4m8s
	BMP	3,174	8	310,909,256	1m21s
		7,462	19	468,273,580	2m35s
Acrobat.exe	BMP	1,440	6	658,370,048	4m25s
		3,678	6	663,923,080	5m2s
	PNG	770	21	297,492,758	3m8s
		1,250	12	354,685,431	4m31s
	JPEG	1,012	13	328,365,912	4m14s
		2,356	4	356,136,453	4m36s

closely related to memory allocation functions. Thus, we measure how many input bytes can flow into memory allocation functions.

Table V shows the results of the evaluation on `ImageMagick`, `Google Picasa`, and `Acrobat`. We input well-formed images (including the PNG, TIF, JPEG, BMP, and GIF formats) to the three applications. Note that well-formed images are obtained either from the Internet (using Google Image Search) or our local disks.

The "Input Format" and "Input Size" columns in Table V represent the format and size of well-formed images, respectively. We measure the trace length and performance overhead, as shown in the two rightmost columns. The performance overhead is acceptable. In most cases, taint analysis takes several minutes for a well-formed image.

We also count the number of hot bytes in well-formed input data, as shown in the "Hot Bytes" column. The size of well-formed inputs is roughly in the range from 1,000 to 7,000 bytes, but the number of hot bytes is less than 50. The reason is that memory allocations during displaying an image usually depend on only a few fields in the image, such as the width, length, and color depth fields.

TaintScope can further modify such bytes to generate malformed inputs. As we can see in Table V, only a small portion of input data are hot bytes. Thus, taint-based fuzzing can dramatically reduce the mutation space.

6.5. Symbolic-Execution-Based Fuzzing Case Study

This section presents a case study that uses the symbolic-execution-based fuzzing to find vulnerabilities in Adobe Flash Player. Our experiment results are encouraging: we detect 3 previously unknown vulnerabilities in Adobe Flash Player.

Adobe claims[18] that Adobe Flash Player is the world's most pervasive software platform reaching 99% of Internet-enabled desktop in markets. Since Adobe Flash Player is heavily-fuzzed and a mature production, it is very hard for traditional fuzzing tools

[18]AdobeFlash. Flash Player penetration. http://www.adobe.com/products/player_census/flashplayer/.

to detect previously unknown vulnerabilities. For instance, we use the MiniFuzz[19] tool from Microsoft to fuzz Adobe Flash Player. After 24 hours, MiniFuzz does not find any vulnerability. Even with taint-based fuzzing, we still do not detect vulnerabilities in Adobe Flash Player. However, our symbolic-execution-based fuzzing successfully discovers 3 previously unknown vulnerabilities in Adobe Flash Player.

The input format of Adobe Flash Player (traditionally called “ShockWave Flash”) SWF[20] is a partially open repository for various text, video, audio and graphic formats. A SWF file consists of a series of tagged data blocks, and the DefineBits tag and several of its variations are used to define image characters with JPEG compression or PNG compression. We find three previously unknown vulnerabilities in the code for parsing the DefineBits tag and its variations. An attacker may exploit these vulnerabilities to execute arbitrary code.

Testing Methodology. First, we choose four PNG images and four JPEG images and use SWFTools[21] to convert them to eight SWF files. TaintScope identifies one checksum check point when Adobe Flash Player processes the SWF files created from PNG images. TaintScope modifies Adobe Flash Player to disable the checksum check. For the SWF files created from JPEG images, TaintScope does not identify checksum checks.

Next, TaintScope runs the modified Adobe Flash Player with these SWF files and logs the execution traces. Then, TaintScope replays each recorded trace, treats the hot bytes as symbolic values and performs offline dynamic symbolic execution. TaintScope detects vulnerabilities using the techniques described in section 4.2. We set a 5-minute timeout for constraint solving. If Z3 cannot response a query in 5 minutes, TaintScope treats the query as unsatisfiable.

In practice, we find that offline dynamic symbolic execution is very slow and the number of instructions in a trace is very large. We do not negate constraints one by one to generate new inputs that can explore different program paths.

Results. TaintScope inspects 8 traces in 24 hours, and each trace consumes roughly 3 hours. However, the results are encouraging. TaintScope successfully discovers three previously unknown vulnerabilities and creates several crashing files.

Figure 4 shows the pseudocode for a new vulnerability we detect (CVE-2010-2170) in Adobe Flash Player. Many other related trace constraints are not shown. Variables `x` and `y` store two 16 bits values from input SWF files. The developers seem to be aware of potential integer overflow problems and enforce several strict checks in line 4 which reject many malformed inputs. In addition, arbitrarily too large `x*y*4` will cause memory allocation failure that can be caught by Adobe Flash Player, resulting in the integer overflow problem in line 5 still invisible. TaintScope successfully detects the vulnerability and generates a crashing file. For example, `x=0xfffa` and `y=0x4002` can trigger the integer overflow issue. In particular, there is a checksum check before the memory allocation. TaintScope repairs the checksum field in the crashing file and confirms the issue.

Table VI reports the statistics over all test runs. The first column presents the average input SWF file size in bytes (the number in parentheses is the mean number of hot bytes). Note that only the hot bytes are treated as symbolic values. The second column shows the average trace length (the number of x86 instructions). Although there are millions of x86 instructions on each trace, only a small portion are symbolically evaluated, as shown in the third column. We also report the average number of VEX

[19]MiniFuzz. File Fuzzer. http://www.microsoft.com/security/sdl/getstarted/tools.aspx.
[20]SWF. The SWF File Format. http://www.adobe.com/devnet/swf/.
[21]SWFTools. SWF manipulation and generation utilities. http://www.swftools.org/.

Table VI. Symbolic-Execution-Based Fuzzing Statistics Results

Input Size	x86 Instrs	Symbolic Instrs	VEX Stmts	Constraints	Time(s)
51,485.6 (16)	95,771,481.2	4,318,044.4	29,739,150.2	357,414.5	9,678.7

IR statements in the fourth column. The fifth column reports the average number of collected trace constraints and the rightmost column presents the mean time spent in offline dynamic symbolic execution for a trace.

Observations. With the growth of the number of initial symbolic inputs, dynamic symbolic execution becomes very slow and the memory consumption increases rapidly. During the propagation of input data, more and more x86 instructions are symbolically executed. In our experiment, if we mark all input bytes as symbolic values, TaintScope aborts after collecting hundreds of thousands of trace conditions because of running out of memory.

The other reason that causes dynamic symbolic execution to be expensive is the wide use of encoded integers in the SWF format. To save space, a 32-bit integer can be encoded in 1–5 bytes depending on the value. A large number of complex operations (such as bitwise AND and XOR) are involved during the decoding process, leading to the exponential growth of symbolic expressions.

Only marking hot bytes as symbolic values can significantly alleviate these problems. Even so, the number of collected trace constraints is very large. First, some hot bytes were used as loop bounds and each loop iteration condition were collected. Second, many nonsymbolic trace conditions are also included. In our current implementation, any expression that includes a symbolic factor is considered a symbolic expression. For example, "x+1-x" is treated as a symbolic value. Although the Z3 solver provides an API `Z3_simplify` to simplify an expression, we find the simplification may cause very heavy time overhead especially for complex symbolic expressions. Instead, we only take the simple but fast heuristic strategy to determine whether an expression is symbolic or not. Thus, the collected trace constraints may include many nonsymbolic predictions.

6.6. Fuzzing Results

As a fuzzing system, TaintScope has already detected 30 severe vulnerabilities in widely used applications and libraries, such as `Microsoft Paint`, `Adobe Acrobat`, `Adobe Flash Player`, `Google Picasa`, `ImageMagick`, and `Libtiff`. Table VII summarizes our results. The "#Vulns" column shows the number of vulnerabilities in each application. If TaintScope locates the checksum checks, it modifies the program to disable them. Otherwise, TaintScope directly performs taint-based fuzzing or symbolic-execution-based fuzzing. The "Checksum-aware" column indicates whether detecting the vulnerabilities requires disabling checksum checks.

We manually analyze the causes of most vulnerabilities and identify five vulnerability types: buffer overflow, integer overflow, double free, null pointer dereference, and infinite loop. According to our vulnerability reports, Secunia[22] and oCERT[23] have confirmed and published security advisories for most of these vulnerabilities and vendors have also released corresponding patches. The "Advisory" column shows the advisory identifier information. *CVE-xxxx*s represent CVE (Common Vulnerabilities and Exposures) identifiers and *SAxxxx*s are security advisories from Secunia. Since a CVE or Secunia identifier may cover multiple vulnerabilities, the number of vulnerabilities in

[22]Secunia. Secunia Web site. http://secunia.com/.
[23]oCERT. Open Source Computer Emergency Response Team. http://www.ocert.org/.

Table VII. Vulnerabilities Detected by TaintScope

Package	Vuln-Type	# Vulns	Checksum-aware?	Advisory	Rating
Microsoft Paint	Int-Overflow	1	N	CVE-2010-0028	Moderate
Google Picasa	Infinite loop	1	N	N/A	N/A
	Int-Overflow	1		SA38435	Moderate
Adobe Acrobat	Infinite loop	1	N	CVE-2009-2995	Extremely
	Int-Overflow	1	N	CVE-2009-2989	Extremely
Adobe Flash Player	Int-Overflow	3	N	CVE-2010-2170	Extremely
			Y	CVE-2010-2171	Extremely
ImageMagick	Int-Overflow	1	N	CVE-2009-1882	Moderate
CamlImage	Int-Overflow	3	**Y**	CVE-2009-2660	Moderate
LibTIFF	Int-Overflow	2	N	CVE-2009-2347	Moderate
wxWidgets	Buf-Overflow	2	N	CVE-2009-2369	Moderate
	Double Free	1	**Y**		
IrfanView	Int-Overflow	1	N	CVE-2009-2118	High
GStreamer	Int-Overflow	1	**Y**	CVE-2009-1932	Moderate
Dillo	Int-Overflow	1	**Y**	CVE-2009-2294	High
XEmacs	Int-Overflow	3	**Y**	CVE-2009-2688	Moderate
	Null-Deref	1	N	N/A	N/A
MPlayer	Null-Deref	2	N	N/A	N/A
PDFlib-lite	Int-Overflow	1	**Y**	SA35180	Moderate
Amaya	Int-Overflow	2	**Y**	SA34531	High
Winamp	Buf-Overflow	1	N	SA35126	High
Total		30			

Table VII does not match the number of identifiers. For example, the identifier CVE-2009-2688 contains three vulnerabilities in XEmacs and the identifier CVE-2010-2171 contains two vulnerabilities in Adobe Flash Player.

The rightmost column shows the Secunia's severity rating for the vulnerabilities. "High" is typically used for remotely exploitable vulnerabilities and "Moderate" is typically used for vulnerabilities that require user interaction. Considering that some vulnerabilities are still in the process of being fixed and/or may be easily exploitable, we do not provide detailed information about these vulnerabilities at this time.

`Adobe Acrobat` can create PDF files from images. TaintScope constructs images (in two different formats) which can cause `Adobe Acrobat` to crash or consume 100% CPU. According to our report and the crashing test case, Secunia has confirmed the memory corruption vulnerability and contacted the vendor. The vendor asked Secunia to postpone the publication of the advisory until a fix is available. We have also confirmed the infinite loop in a binary file named `ImageConversion.api` and already sent the PoC (proof of concept) image to Adobe PSIRT (Adobe Product Security Incident Response Team). Adobe has published a security bulletin for the two vulnerabilities and released patches.[24]

`Microsoft Paint` has been included in all versions of Microsoft Windows. TaintScope discovered an integer overflow in gdiplus.dll which causes an erroneous memory allocation in the `Paint` program when `Paint` opens a malformed JPEG image.

[24]Apsb09-15. Adobe Security Bulletins. http://www.adobe.com/support/security/bulletins/apsb09-15.html.

```
509 wxPNGHandler::LoadFile(wxImage *image,
...//ignore the CRC checks
575 for (i = 0; i < height; i++){
577   if ((lines[i]=(unsigned char*)malloc(width*4))
          == NULL){
579     for ( unsigned int n = 0; n < i; n++ )
580       free( lines[n] ); //first time free()
581     goto error;
...
621 error:
...
630  if ( lines )
631  {
632    for ( unsigned int n = 0; n < height; n++ )
633      free( lines[n] ); //second time free()
```

Fig. 5. A double free vulnerability in wxPNGHandler::LoadFile() in wxWidgets 2.8.10.

Successful exploitation may allow execution of arbitrary code. Microsoft has published a security bulletin MS10-005[25] according to our report.

TaintScope also discovers an integer overflow and an infinite loop in `Google Picasa`. We first sent the infinite loop information to the Google Security Team, however, after the initial contact, we have not heard back from Google for months. Recently, we found the infinite loop issue has been fixed in its new version. The integer overflow occurs in PicasaPhotoViewer.exe when processing JPEG files, which finally causes a heap buffer overflow.

Many detected vulnerabilities in Table VII require disabling checksum checks. As an example, we present the double free vulnerability[26] in `wxWidgets`, a popular open source cross-platform GUI toolkit. The double free vulnerability can be exploited to potentially execute arbitrary code via a specially crafted PNG file. A basic functionality of `wxWidgets` is to load images in a variety of formats. Specifically, the "`wxPNGHandler::LoadFile()`" function in `wxWidgets` is responsible for loading an image in the PNG format. This function first checks the CRC checksum values in a PNG image, and then processes the image data row by row. Figure 5 shows the source code snippet of the function. Note that `lines[n]` may be freed twice at lines 580 and 633 if the function `malloc` at line 577 returns a NULL pointer.

TaintScope accurately locates the CRC checksum check points in `wxWidgets` and the checksum fields in PNG images. Meanwhile, TaintScope identifies the hot bytes in well-formed PNG images which can affect the variable "`width`" used in line 577. In fact, the variable "`width`" corresponds to the width field in a PNG image. At the fuzzing phase, TaintScope modifies the hot bytes to some extremal values and alters the execution flows at the checksum check points. Several malformed test cases trigger the double free vulnerability in the function `wxPNGHandler::LoadFile()`. At the phase of repairing test cases, TaintScope first records the execution trace with a malformed PNG image, and reexecutes the trace using dynamic symbolic execution. By solving the trace constraints, TaintScope successfully generates a valid checksum value for the malformed PNG image. The new PNG image can pass initial checksum checks in `wxWidgets` and trigger the double-free vulnerability. Since the width field in a PNG image is protected by checksum values, we believe the vulnerability cannot be detected by random modification.

For more details on the published vulnerabilities in Table VII, we refer the readers to Secunia[27]. For instance, Secunia has developed exploits and PoC code for the

[25]MS10-005. Microsoft security bulletin.
http://www.microsoft.com/technet/security/Bulletin/25MS10-005.mspx.
[26]SA35292. wxWidgets Double Free Vulnerabilities. http://secunia.com/advisories/35292/.
[27]See footnote 22.

vulnerabilities in `Gstreamer`[28] and `Winamp`[29] (only available for certain types of vendors and governments).

In summary, our experiments show the following.

— TaintScope can accurately locate the checksum check points in programs and perform checksum-aware fuzzing.
— Only a small portion of input data are hot bytes. Thus, taint-based fuzzing can dramatically reduce the mutation space.
— While taint-based fuzzing provides a quick test on target programs, symbolic-execution-based fuzzing carefully inspects an execution trace and nicely complements taint-based fuzzing.
— Dynamic symbolic execution is slow, but it is powerful. Based on offline dynamic symbolic execution, TaintScope can find subtle vulnerabilities and fix checksum fields.
— TaintScope successfully detects dozens of vulnerabilities in real-world programs.

7. DISCUSSION

In this section, we discuss the limitations in the current implementation of TaintScope.

First, TaintScope currently cannot deal with secure integrity check schemes, such as keyed cryptographic hash algorithms or digital signature, which are designed to protect against intentional data alteration. Although TaintScope can locate and bypass such checks at the checksum-aware fuzzing phase, it is impossible for TaintScope to automatically generate test cases with valid digital signatures. From software testing point of view, some vulnerabilities could be hidden behind such complex application defenses (e.g., digital signatures). We suggest the software developers disable such defense mechanisms at testing phase. We leave the full study of this problem as future work.

Second, the effectiveness of TaintScope may be limited when all input data are encrypted. After data decryption, the complex data dependency relationships will heavily influence hot bytes detection and checksum identification. A possible mitigation strategy is to configure TaintScope to track the decrypted data only. Recent research such as ReFormat [Wang et al. 2009b], Dispatcher [Caballero et al. 2009] and BitFuzz [Caballero et al. 2010] already shows some promising results to locate encryption/decryption routines. Such techniques could be combined with TaintScope to detect the vulnerabilities when the target program processes the decrypted data.

Third, a program may utilize multi-layered checksum checks. In this case, we could iteratively apply TaintScope to locate the checksum check points one by one. We leave this as future work.

In addition, due to the complexity of the x86 instruction set, the current taint analysis in TaintScope does not instrument all kinds of x86 instructions. TaintScope also ignores the control flow dependencies. However, previous work [Clause and Orso 2009] reveals that tracking control flow dependencies may make too much noise. Extending TaintScope to track control flow propagation (similar to [Egele et al. 2007] and DYTAN [Clause et al. 2007]) and improving data flow propagation are left as future work.

8. RELATED WORK

Traditional Fuzzing. Miller et al. [1990] first proposed the concept of fuzzing. Their simple tool generated streams of random characters, sent them to target programs,

[28]SA35205. GStreamer Good Plug-ins PNG Processing Integer Overflow Vulnerability. http://secunia.com/advisories/35205/.
[29]SA35126. Winamp MP4 Processing Buffer Overflow Vulnerabilities. http://secunia.com/advisories/35126/.

and was able to crash 25-33% of the utility programs on UNIX systems. Since then, a wide range of fuzzing tools have been developed. Sutton et al. [2007] presented an overview of fuzzing techniques and tools. Traditionally, there are two main ways to get malformed inputs: *data generation* and *data mutation* [Oehlert 2005]. The former (e.g., Spike[30], Peach[31], SNOOZE [Banks et al. 2006]) generates malformed inputs based on predefined specifications and the latter (e.g., FileFuzz[32]) mutates well-formed inputs. However, the common drawback of traditional fuzzing techniques is that most malformed inputs are prematurely dropped. To improve the effectiveness of fuzzing tools, the following two categories of new techniques are proposed.

Symbolic-execution-based white-box fuzzing/unit testing. This technique has been widely implemented in numerous tools, such as CUTE [Sen et al. 2005], DART [Godefroid et al. 2005], SAGE [Godefroid et al. 2008a], SmartFuzz [Molnar et al. 2009], EXE [Cadar et al. 2008], and KLEE [Cadar et al. 2008]. In general, based on code instrumentation or program tracing, these tools replace concrete input data with symbolic values, collect and solve the constraints on execution traces and guide input error detection and generation. When testing applications with highly-structured inputs, such as compilers and interpreters, Godefroid et al. [2008b] and Majumdar and Xu [2007] proposed a variation technique which employs input symbolic grammar specifications. These tools have proven to highly improve the effectiveness of traditional fuzzing tools. They successfully detected serious bugs in GNU coreutils [Cadar et al. 2008], large shipped Windows applications [Godefroid et al. 2008a; 2008b], and Linux file systems [Cadar et al. 2008].

Automatic generation of test cases with correct checksum fields but without knowing checksum algorithm is a great challenge. For simple checksum algorithms such as integer addition, existing dynamic symbolic execution systems such as Replayer [Newsome et al. 2006] are able to automatically generate correct checksums. However, as shown in Newsome et al. [2006], the checksum computation significantly increases the complexity of the collected symbolic formula. Furthermore, previous studies [Brumley et al. 2008; Sharif et al. 2008] have revealed that current dynamic symbolic execution engines and constraint solvers cannot accurately generate and solve the constraints that describe the complete process of complex checksum algorithms. Newsome et al. [2006] also proposed a reasonable scheme to deal with complex checksum algorithms, that is, *iteratively* constraining some of the input variables to have the concrete values and then simplifying and solving the trace constraints. Instead, TaintScope *directly* leaves all bytes concrete except those in the checksum fields, which significantly reduces the complexity of the trace constraints.

The closest work to ours is BitFuzz [Caballero et al. 2010]. Besides the checksum problem, BitFuzz can also deal with some encoding functions (such as decryption and decompression) if BitFuzz can identify their inverse functions (such as encryption and compression). While BitFuzz focuses on test case generation to explore the execution space, TaintScope uses offline dynamic symbolic execution to detect vulnerabilities on a single trace. In the future, we will also utilize BitFuzz's approach to overcome the encoding function problems.

Taint-analysis-based fuzzing. BuzzFuzz [Ganesh et al. 2009] is a taint-based fuzzing tool. To track taint information, BuzzFuzz needs to instrument an application's *source code*. The instrumented code is responsible for identifying input data that can

[30]See footnote 1.
[31]See footnote 2.
[32]FileFuzz. From iDefense Labs: http://www.labs.idefense.com/software/fuzzing.php.

influence the values at system calls. However, modern applications make extensive use of third-party libraries. BuzzFuzz cannot instrument such libraries if source code is unavailable, leading to the loss of taint information. In contrast, TaintScope directly works on binary executables and can monitor the execution of all libraries. In addition, TaintScope can bypass checksum checks in programs.

Flayer [Drewry and Ormandy 2007] is a taint tracing tool with the ability to redirect the execution flows through the modification of conditional jump instructions. Flayer is based upon functionality from Memcheck [Nethercote and Seward 2007] and marks input data only with 0/1 label. Thus, Flayer cannot accurately track the impact of input data on an application's execution. Execution flow alteration is also used in analyzing malware behavior, for instance, exploring multiple execution paths [Moser et al. 2007], and forcing sampled execution to identify various kernel rootkit behaviors [Wilhelm and Chiueh 2007].

Corpus Distillation[33] is a feedback-driven fuzzing system that uses a code coverage heuristic to select and mutate input samples. In particular, to overcome checksum problems, Corpus Distillation designed the subinstruction profiling method, that is, comparison instructions (such as immediate comparison and `rep cmps`) are broken into bit-sized chunks and the coverage score of these instructions depends on the "depth" of comparison. Based on the subinstruction profiling, Corpus Distillation is able to generate correct CRC checksums in PNG files without the requirement of constraint solving. However, due to the lack of fine-grained taint tracking and checksum field identification, to pass the checksum checks, Corpus Distillation has to mutate all bits in an input sample until it reaches checksum fields, which may heavily limit its efficiency.

Moreover, Corpus Distillation and TaintScope can benefit from each other. While TaintScope's checksum check points locating and bypassing techniques can be used in Corpus Distillation to improve the sub-instruction profiling method, the code-coverage-based input sample selection and mutation technique in Corpus Distillation can also be used in the fuzzing phase of TaintScope.

Many protocol reverse-engineering tools (such as Prospex [Comparetti et al. 2009], Tupni [Cui et al. 2008], AutoFormat [Lin et al. 2008], Polyglot [Caballero et al. 2007], Discoverer [Cui et al. 2007]) can be used to guide fuzzing tests. These tools can extract format specifications for input data by analyzing network traffic, monitoring the execution of a program while it processes input data, or analyzing binary executables directly. The extracted protocol specifications can be further translated into fuzzing specifications. However, none of these systems explicitly discussed how to bypass checksum checks.

In addition, Kang et al. [2009] proposed a trace matching algorithm to locate the divergence point between two similar traces. This algorithm could also potentially be applied to locate checksum check points in programs. Johnson et al. [2011] proposed differential slicing method, which can be used to diagnose why malformed inputs cause the program to crash and analyze the fuzzing results.

Finally, there are a significant amount of vulnerability detection studies based on static analysis. We refer the readers to StaticAnalysis[34] for further references. In particular, [Wang et al. 2009b] and [Cova et al. 2006] are two binary analysis tools which can identify integer overflow vulnerabilities or insecure uses of sensitive C library calls in binary executables. While TaintScope is mainly a dynamic fuzzing tool, it could also benefit from the advances in this line of work. For example, TaintScope uses similar

[33]Making Software Dumber. http://taviso.decsystem.org/making_software_dumber.pdf.
[34]StaticAnalysis. List of tools for static code analysis.
http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

heuristics in [Wang et al. 2009b] to specially monitor the size arguments of memory allocation functions and has discovered many integer overflow bugs.

9. CONCLUSIONS

This article presents TaintScope. First, TaintScope can locate checksum-based integrity checks in programs, then enforce execution flow alterations at located checksum check points to make malformed input to pass the checksum checks. Second, TaintScope has two ways to produce malformed inputs, taint-based fuzzing and symbolic-execution-based fuzzing. Taint-based fuzzing focuses on modifying hot bytes in a well-formed instance, which dramatically reduces the mutation space. Symbolic-execution-based fuzzing uses offline dynamic symbolic execution to collect trace constraints on hot bytes, which can find subtle vulnerabilities on the trace. Finally, TaintScope can automatically fix the checksum fields in malformed test cases using combined concrete and symbolic execution.

We apply TaintScope to 19 real-world applications. Experimental results show that TaintScope can accurately locate the checksum checks in programs and dramatically improve the effectiveness of fuzz testing. TaintScope has already identified 30 previously unknown vulnerabilities in several widely used applications.

REFERENCES

Balakrishnan, G. and Reps, T. 2010. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst. 32*, 23:1–23:84.

Banks, G., Cova, M., Felmetsger, V., Almeroth, K., Kemmerer, R., and Vigna, G. 2006. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *Proceedings of the Information Security Conference (ISC)*. Springer.

Boutell, T. 1997. PNG Specification. RFC 2083, Internet Engineering Task Force.

Brumley, D., Hartwig, C., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., Song, D., and Yin, H. 2007. Bitscope: Automatically dissecting malicious binaries. Tech. rep. CMU-CS-07-133, Carnegie Mellon University.

Brumley, D., Wang, H., Jha, S., and Song, D. 2007. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF'07)*. IEEE Computer Society, Los Alamitos, CA, 311–325.

Brumley, D., Poosankam, P., Song, D., and Zheng, J. 2008. Automatic patch-based exploit generation is possible: Techniques and implications. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'08)*. IEEE Computer Society, Los Alamitos, CA, 143–157.

Caballero, J., Yin, H., Liang, Z., and Song, D. 2007. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 317–329.

Caballero, J., Poosankam, P., Kreibich, C., and Song, D. 2009. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, New York, NY, 621–634.

Caballero, J., Poosankam, P., McCamant, S., Babic, D., and Song, D. 2010. Input generation via decomposition and re-stitching: Finding bugs in malware. In *Proceedings of the 17th ACM Conference on Computer and Communication Security*. ACM, New York. NY.

Cadar, C., Dunbar, D., and Engler, D. 2008. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, 209–224.

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., and Engler, D. R. 2008. Exe: Automatically generating inputs of death. *ACM Trans. Info. Syst. Sec. 12*, 2, 1–38.

Cifuentes, C. 1994. Reverse compilation techniques. Ph.D. thesis, Queensland University of Technology, Australia.

Clause, J. and Orso, A. 2009. Penumbra: Automatically identifying failure-relevant inputs using dynamic tainting. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*. ACM, New York, NY, 249–260.

CLAUSE, J., LI, W., AND ORSO, A. 2007. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'07)*. ACM, New York, NY, 196–206.

COMPARETTI, P. M., WONDRACEK, G., KRUEGEL, C., AND KIRDA, E. 2009. Prospex: Protocol specification extraction. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (SP'09)*. IEEE Computer Society, Los Alamitos, CA, 110–125.

COVA, M., FELMETSGER, V., BANKS, G., AND VIGNA, G. 2006. Static detection of vulnerabilities in x86 executables. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE Computer Society, Los Alamitos, CA, 269–278.

CUI, W., KANNAN, J., AND WANG, H. J. 2007. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of 16th USENIX Security Symposium (SS'07)*. USENIX Association, Berkeley, CA, 1–14.

CUI, W., PEINADO, M., CHEN, K., WANG, H. J., AND IRUN-BRIZ, L. 2008. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*. ACM, New York, NY, 391–402.

DEUTSCH, P. AND GAILLY, J.-L. 1996. ZLIB compressed data format specification version 3.3. RFC 1950, Internet Engineering Task Force.

DREWRY, W. AND ORMANDY, T. 2007. Flayer: Exposing application internals. In *Proceedings of the 1st USENIX Workshop on Offensive Technologies (WOOT'07)*. USENIX Association, Berkeley, CA, 1–9.

EGELE, M., KRUEGEL, C., KIRDA, E., YIN, H., AND SONG, D. 2007. Dynamic spyware analysis. In *Proceedings of the USENIX Annual Technical Conference (ATC'07)*. USENIX Association, Berkeley, CA, 1–14.

ELKARABLIEH, B., GODEFROID, P., AND LEVIN, M. Y. 2009. Precise pointer reasoning for dynamic test generation. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA'09)*. ACM, New York, NY, 129–140.

GANESH, V., LEEK, T., AND RINARD, M. 2009. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. IEEE Computer Society, Los Alamitos, CA, 474–484.

GODEFROID, P., KLARLUND, N., AND SEN, K. 2005. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 213–223.

GODEFROID, P., KIEZUN, A., AND LEVIN, M. Y. 2008a. Grammar-based whitebox fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY, 206–215.

GODEFROID, P., LEVIN, M., AND MOLNAR, D. 2008b. Automated whitebox fuzz testing. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*.

JOHNSON, N., CABALLERO, J., CHEN, K. Z., MCCAMANT, S., POOSANKAM, P., REYNAUD, D., AND SONG, D. 2011. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland'11)*.

KANG, M. G., YIN, H., HANNA, S., MCCAMANT, S., AND SONG, D. 2009. Emulating emulation-resistant malware. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec)*.

KORN, D., MACDONALD, J., MOGUL, J., AND VO, K. 2002. The VCDIFF generic differencing and compression data format. RFC 3284, Internet Engineering Task Force.

LIN, Z., JIANG, X., XU, D., AND ZHANG, X. 2008. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*.

LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, 190–200.

MAJUMDAR, R. AND XU, R.-G. 2007. Directed test generation using symbolic grammars. In *Proceedings of the 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE Companion'07)*. ACM, New York, NY, 553–556.

MILLER, B. P., FREDRIKSEN, L., AND BRYAN, S. 1990. An empirical study of the reliability of UNIX utilities. *Comm. ACM 33*, 12, 32–44.

MOBB. 2006. http://browserfun.blogspot.com.

MOKB. 2006. Month of Kernel Bugs. http://projects.info-pull.com/mokb/.

MOLNAR, D., LI, X. C., AND WAGNER, D. A. 2009. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th USENIX Security Symposium*.

MOSER, A., KRUEGEL, C., AND KIRDA, E. 2007. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'07)*. IEEE Computer Society, Los Alamitos, CA, 231–245.

NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 89–100.

NEWSOME, J., BRUMLEY, D., FRANKLIN, J., AND SONG, D. 2006. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS'06)*. ACM, New York, NY, 311–321.

OEHLERT, P. 2005. Violating assumptions with fuzzing. *IEEE Sec. Priv. 3*, 2, 58–62.

POSTEL, J. 1981. Internet protocol. RFC 791, Internet Engineering Task Force.

SEN, K., MARINOV, D., AND AGHA, G. 2005. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*. ACM, New York, NY, 263–272.

SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. 2008. Impeding malware analysis using conditional code obfuscation. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*.

STALLINGS, W. 2005. *Cryptography and Network Security* 4th Ed. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

SUTTON, M., GREENE, A., AND AMINI, P. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

WANG, T., WEI, T., LIN, Z., AND ZOU, W. 2009a. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In *Proceedings of the 16th Annual Network and Distributed System Security Symposium*.

WANG, Z., JIANG, X., CUI, W., WANG, X., AND GRACE, M. 2009b. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of the 14th European Conference on Research in Computer Security (ESORICS'09)*. Springer-Verlag, Berlin, 200–215.

WEI, T., MAO, J., ZOU, W., AND CHEN, Y. 2007. Structuring 2-way branches in binary executables. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC'07)*. IEEE Computer Society, Los Alamitos, CA, 115–118.

WILHELM, J. AND CHIUEH, T.-C. 2007. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection (RAID'07)*. Springer-Verlag, Berlin, 219–235.

WONDRACEK, G., MILANI COMPARETTI, P., KRUEGEL, C., AND KIRDA, E. 2008. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*.

ZALEWSKI, M. 2007. Bunny-the-fuzzer: Instrumented c code security fuzzer. http://code.google.com/p/bunny-the-fuzzer/.

Received August 2010; revised January 2011, March 2011; accepted April 2011