Analysis of Exception-Based Control Transfers

Babak Yadegari
University of Arizona
Tucson, AZ
babaky@cs.arizona.edu

Jon Stephens
University of Arizona
Tucson, AZ
stephensj2@cs.arizona.edu

Saumya Debray
University of Arizona
Tucson, AZ
debray@cs.arizona.edu

Abstract

Dynamic taint analysis and symbolic execution find many important applications in security-related program analyses. However, current techniques for such analyses do not take proper account of control transfers due to exceptions. As a result, they can fail to account for implicit flows arising from exception-based control transfers, leading to loss of precision and potential false negatives in analysis results. While the idea of using exceptions for obfuscating (unconditional) control transfers is well known, we are not aware of any prior work discussing the use of exceptions to implement conditional control transfers and implicit information flows. This paper demonstrates the problems that can arise in existing dynamic taint analysis and symbolic execution systems due to exception-based implicit information flows and proposes a generic architecture-agnostic solution for reasoning about the behavior of code using user-defined exception handlers. Experimental results from a prototype implementation indicate that the ideas described produce better results than current state-of-the-art systems.

1. INTRODUCTION

Dynamic taint analysis and symbolic execution find numerous applications in privacy- and security-related program analyses. Dynamic taint analysis involves tracking certain data (the tainted values) through the execution of the program; symbolic execution generates path constraints that express the logic of the computation along an execution path. The two techniques can be combined to reason about how input values influence the execution path taken by the program, and thereby identify alternative inputs that can cause a different execution path to be taken. This approach has been applied to a wide variety security and software testing applications, e.g., test case and exploit generation [7, 8, 14, 24, 46], vulnerability detection [9, 10, 16], and multi-path exploration during dynamic analysis of malware code [2, 3, 6, 35, 36].

Given the numerous security-related applications of dynamic taint analysis and symbolic execution, it is important to understand any potential shortcomings of current techniques and devise solutions to mitigate their weaknesses; a failure to do so can potentially lead to flawed conclusions about the software we analyze, e.g., leave application vulnerabilities unidentified or malware execution paths unexplored. Previous studies have discussed some of the limitations of dynamic taint analysis [13,44] and symbolic execution [48,56]. These works all involve "normal" executions where the execution path is (at least in principle) available for inspection. This paper focuses on a different kind of control flow construct, namely, exception-based control transfers, where the execution path is not explicit and which can be significantly harder to analyze than conventional control transfers.

It turns out that exception-based control transfers—which are well-known as an obfuscation mechanism and have been widely used in malicious code—can be used to realize implicit information flows in ways that are not detected by any of the existing state-of-the-art dynamic analysis tools we tested including KLEE [7], S2E [16], FuzzBall [4] and angr [49]. The problem arises both for dynamic taint analysis and for symbolic execution. This is because currently, implicit information flow detection and symbolic execution techniques rely on information about the execution path and control flow of the program to reason about alternative execution paths. For instance, dynamic taint analysis approaches (e.g. [17,29]) use program's control flow graph (CFG) to determine whether a dataflow is implicit, i.e., occurs through a control transfer. Similarly in symbolic execution, the analysis relies on syntactic characteristics such as instruction opcodes to identify jump instructions and thereby determine branch conditions and path constraints.

On the other hand, operating systems provide powerful low-level mechanisms, namely, *exceptions*, that allow programs to deal with unexpected situations. These are hardware or software faults (e.g. memory access violation) that cause execution to be transferred to *exception handlers* that take appropriate action. The handlers may be custom routines implemented within an application or default handlers provided by the operating system, and the details of the exception-handling mechanism may depend on the underlying operating system, but at the end, they all have the common behavior of transferring the control of application to a handler that is responsible for handling the occurred exception or fault. The control flow behavior of a program can be obfuscated by using deliberately-induced exceptions to cause execution to be transferred to the handler code as part of the program's "normal" execution [31, 40, 57]. The simplest manifestation of this idea uses exceptions to implement unconditional jumps

CODASPY '17, March 22–24, 2017, Scottsdale, AZ, USA.

DOI: http://dx.doi.org/10.1145/3029806.3029826

to the handler code; however, it is straightforward to extend this idea such that the exception is raised only under specific conditions, thereby realizing exception-based conditional control transfers.

Current techniques for dynamic taint analysis and symbolic execution only take into account normal control transfers and do not consider control transfers due to exceptions. This means that an attacker can exploit the exception mechanism to implement arbitrary control transfers and thereby realize implicit information flows that are not detected using existing analysis techniques. This represents a significant shortcoming of such analyses. Furthermore, fixing the problem is not straightforward because identifying the potential control transfer—i.e., the locations where exceptions can be raised and those where they may be handled—is not always straightforward. We have observed such information flows in existing exploit code. While symbolic execution has a number of challenges, such as path explosion, constraint solving, system calls etc., these have been previously discussed in the literature [11, 45] and so are not pursued further here. This paper focuses on a different kind of challenge for symbolic execution that has not been previously studied in detail in the research literature.

This paper proposes a solution to excpetion-based control transers by taking into account the possiblity of a control transfer due to exceptions where the application code contains user-defined exception handlers. In particular, it makes the following contributions:

- It demonstrates the challenges that arise in current dynamic taint analysis and symbolic execution techniques when dealing with exception-based information flow.
- It proposes a generic architecture-agnostic solution for both dynamic taint analysis and symbolic execution to address the shortcomings and hence improves the accuracy and robustness of the analyses.
- It describes a prototype implementation that outperforms the state-of-the-art systems for symbolic execution and dynamic taint analysis if applied against real mal-intended code as well as sample test programs that use exception-based obfuscation techniques.

The remainder of the paper is organized as follows: section 2 discusses motivations and background on dynamic taint analysis and symbolic execution. Section 3 demonstrates the problem in more detail and discusses our ideas to solve this problem followed by our prototype implementation details in section 4. Our evaluation results are presented in section 5. Previous and related works discussed in section 6 and the paper concludes in section 7.

2. BACKGROUND AND MOTIVATING EXAMPLE

2.1 Signals

Signals (exceptions) serve to notify processes of an event or a software or hardware fault. There are two types of signals: *synchronous* and *asynchronous*. A synchronous signal typically results from an error in executing an instruction, e.g., "illegal instruction" (`SIGILL`) or "illegal memory access" (`SIGSEGV`). Asynchronous signals are delivered asynchronously and do not depend on the execution context, e.g., `SIGSTOP`, which stops execution of the receiving process, or `SIGKILL`, which terminates it. Modern operating systems provide APIs to user applications to overwrite default exception handler routines with their own code. When a signal is delivered to the process, control is transferred to the handler code. This gives the processes flexibility to provide their own routines to recover or clean up after unrecoverable faults. For instance, Windows uses a mechanism called *Structured Exception Handling (SEH)* and Linux uses *Signals*. Despite different implementation details, the high-level concepts are similar between different operating systems.

Synchronous signals are triggered in the course of execution of instructions if the hardware or the operating system is not able to successfully execute the instruction. Such errors are usually severe enough that the execution of the process that caused the fault can not be continued. This type of exceptions are common among different operating systems. Asynchronous signals, on the other hand, are specific to the Unix/Linux operating systems and can interrupt the execution of a process at any instructions. These signals are primarily used for synchronization purposes between different or parent/child processes. Since each category is fundamentally different from the other, we have different solutions for each of these types of events in the programs that are discussed in the rest of this section.

2.2 A Motivating Example

The mechanism allowing users to define their own signal handlers can be misused by attackers to obfuscate branch points in the code to confuse analysis. Figure 1 shows two sample programs using this technique to implicitly propagate sensitive data (Figure 1a), and to redirect the flow of execution on certain environments (Figure 1b).

Synchronous Signals.

The code in 1(a) sets handlers for both `SIGSEGV` and `SIGFPE` signals where the former is raised by OS and sent to the process on memory access violations and the latter is raised in case of arithmetic errors such as divide by zero. After copying the sensitive data to `secret` value at line 13, the code intentionally dereferences a `NULL` pointer which raises a `SIGSEGV` and so transfers control to `segv_handler`. After the `segv_handler` function returns, the control transfers to the instruction that originally generated the fault and since the problem has not been fixed, the operating system raises another `SIGSEGV` signal and so on. This continues until variable `c` equals zero meaning that variable `public` is equal to `secret` at line 7. At this point, the division operation line 8 generates an arithmetic fault (divide by zero) which then transfers the control to the `fpe_handler`. Doing so, the attacker is able to copy the `secret` value into another variable without any explicit or direct data flow or implicit flow since the control transfer edges are not part of the static CFG of the program.

Asynchronous Signals.

Figure 1(b) shows a timing technique that is quite frequently used by malicious codes simply as a timeout mechanism. Figure 1(b) shows a variation of the technique where the signal mechanism can be used as an anti-analysis technique. The process sets a handler for `SIGALRM` signal at line 6 where the handler code changes the function pointer `foo` to point to some malicious code. Line 7 registers a wake up

```
1:   int public, secret;
2:   void fpe_handler(){
3:       /* public is equal
                to secret here! */
4:   }
5:   void segv_handler(){
6:       public++;
7:       int c = secret-public;
8:       c = c/c;
9:   }
10:  int main(){
11:      signal(SIGFPE,  fpe_handler);
12:      signal(SIGSEGV, segv_handler);
13:      secret  = get_secret();
14:      char *p = NULL
15:      *p++;
16:  }
```

(a) Using synchronous signals for implicit flow

```
  /* foo is a function pointer */
1:   foo = /* some malicious code */
2:   void alarm_handler(){
3:       foo = /* some benign code */
4:   }
5:   int main(){
6:       signal(SIGALRM, alarm_handler);
7:       alarm(1);
8:       for (int i=0; i < THRESHOLD; i++){
             /* an empty loop that takes more than
             1 second in a hostile environment */
9:       }
10:      alarm(0);
11:      foo();
12:  }
```

(b) Using asynchronous signals to obfuscate control transfers

Figure 1: Branch obfuscation using signals

call being delivered to the process by the operating system in one second. There is second call to `alarm` system call (line 11) which clears pending alarm signals, if there is any, and does not have any effect if the signal is already delivered. This means if the code at lines $8-9$ is executed before the signal arrives, i.e. in less than a second, the process continues and the handler is never executed. Otherwise, the delivered signal causes the exception handler to be called which changes the behaviour of the code. An attacker can put a code in lines $8-9$ that takes less than one second to execute on normal hosts, and takes more than one second in an analysis environment or a "hostile" machine. This results in the malicious part not being executed if the code is being monitored. This behavior is similar to timing attacks used in malicious codes to detect analysis environments and evade detection [32, 42]. Similar to the code in Figure 1(a), there is no obvious control flow edge in the code to guide the symbolic execution to different execution paths. In fact, being able to automatically handle this case is even more subtle than the previous example since the `SIGALRM` is raised asynchronously by the operating system meaning that the *possible* control transfer edge to `alarm_handler` function could be anywhere between the two `alarm` system calls. At what point in the execution the process receives the signal depends on many variables (with `CPU` cycles being the most important one) making the problem non-trivially hard.

2.3 Dynamic Taint Analysis

Dynamic taint analysis involves tracking the flow of certain marked (tainted) data throughout the program execution. This analysis has numerous applications in different areas such as in program debugging and software testing and application security. In the context of security, dynamic taint analysis has led to many successful implementation of security tools that protect applications against a wide range of attacks including buffer overruns [30, 38], SQL injection attacks [25, 39] and formatted string attacks [38, 41].

A potential problem for dynamic taint analysis is that of *under-tainting* caused by not propagating taint through control dependencies. Basically a tainted control transfer results in different execution paths depending on the tainted data which will lead to different data flow equations. To address this problem, researches have tried to combine dynamic data flow analysis with static control flow graphs and propagate taint along the control flow edges [17, 29]. The problem with these approaches is that a static CFG does not capture possible dynamic execution paths that are cause by unexpected events and errors. As showed earlier, this could cause imprecision for the analysis that are sensitive to control flow information, including dynamic taint analysis, and can be used by attackers to evade currently used techniques.

2.4 Symbolic Execution

Symbolic execution consists of executing programs on symbolic variables rather than concrete data which allows the analysis to represent the program in terms of logical and arithmetic formulas. Combined with dynamic taint analysis, symbolic execution can represent program execution path with formulas and constraints when the inputs to the program are marked as symbolic. Using an SMT solver (e.g. [20, 22]), one can solve the path constraints to find alternative input values that cause different paths to be executed. This turns out to be useful for numerous security applications such as automatic vulnerability and exploit generation [7, 14] and malware analysis [6, 36]. However, vanilla symbolic execution relies on explicit control flow constructs that might be obscured or obfuscated when dealing with malicious codes as shown in Figure 1(b). Yadegari *et. al.* [56] mentions some of the challenges of symbolic execution when applied to obfuscated code. In this paper we are showing some other obfuscation techniques that use irregular control flow transfers and are already being used in malicious or even legitimate programs causing problems for symbolic execution.

3. ANALYZING EXECPTION BEHAVIOR

This section discusses our proposed solution to address the analysis problems arising from exception-based control transfers.

3.1 Synchronous Events

As discussed previously, to be able to detect implicit information flow propagation, dynamic taint analysis relies on the static CFG of the program to propagate tainted data along control transfer edges in the CFG. If a location (variable) x is defined by an instruction J that is control dependent on a control transfer instruction I, then x inherits the taint marks of the variable(s) used by the control transfer instruction I;

Schwartz *et al.* refer to this as *control-flow taint* [45]. Correct handling of control-flow taint requires a static CFG that reflects the possible control transfers of the program. A key problem in dealing with exception-based control transfers is that CFGs constructed using conventional techniques typically do not account for exception-based control transfers, potentially leading to under-tainting.

3.1.1 Control Flow Graph Augmentation

A natural approach to fixing the problem of missing control flow edges in the static CFG of the program would be to insert additional edges corresponding to exception-based control transfers. However, a naive solution that statically includes every possible exception-based control transfer edge, from any instruction or statement that can possibly raise an exception, can become so large and cluttered as to be unusable.

Our solution charts a middle ground where we augment a conventional static CFG with additional control transfer edges that are added at runtime as the program executes and more information is available. We add control transfers from instructions that potentially can raise exceptions to the appropriate exception handler. Given a program P and input $\bar{x}$, we do this as follows. Here, S is a mapping that maps different exceptions to (the address of) the handlers that have been registered for them.

1. Static analysis phase:
Construct the static CFG for P.
2. Dynamic analysis phase:

1. Initialize the exception-handler mapping S to $\emptyset$.
2. Execute P on the given input $\bar{x}$. For each instruction I of this execution, do:
   (a) If I is a call to register an exception handler H at address A_H for an exception e, update the exception-handler mapping S to map the exception e to A_H.
   (b) Otherwise: for each exception e for which a handler is known to exist in S, use symbolic execution to determine whether there exists an input that can cause instruction I to raise exception e. If there exists such an input, add a control flow edge from I to the handler $S[e]$.

Identifying whether an instruction raises an exception or not needs run-time information, such as virtual memory pages associated to the process, that are not available statically. Hence the static CFG needs to be modified or augmented at run-time to contain necessary control flow information that are not available otherwise.

Figure 2 shows this transformation on the CFG of the sample code in Figure 1(a). Figure 2(a) shows the original CFG that is built using standard definitions of control flow graph constructions. It can be seen from the CFG, all three functions are isolated and there is no explicit control transfer edge between them. This is because there is not a control flow statement that targets another function from anyone else. This is also true for the machine code generated by the compiler and there is not explicit control flow edge between functions in the compiled code as well. However, as shown in Figure 2(b), there are control transfers between different functions at run-time. The reason is that functions `segv_handler` and `fpe_handler` are registered as signal handlers for `SIGSEGV` and `SIGFPE` signals respectively which cause the control flow being transferred to these functions, should there be any memory or arithmetic errors. The control flow edge from `main` to `segv_handler` is because a `NULL` pointer dereference at `*p++` statement in the `main` function where the control returns back to the same statement after it executes. Similarly, there is a control flow edge from statement `c = c/c` in the segv_handler function to `fpe_handler` that is due to a division by zero if variable `secret` is equal to the `public` in the `segv_handler` function. In fact, this logic implements a loop without any explicit control transfer statement where the code in function `segv_handler` is executed until the variable `secret` is copied to `public` and then exits to `fpe_handler` function.

After creating the augmented CFG, it needs to be analyzed to produce post-dominator information necessary for dynamic taint analysis. Basically, those non-control flow instructions that have an outgoing edge to an exception handler can be treated as conditional jumps for the purpose of control dependence and post-dominator analyses. The following section discusses how we identify the instructions that have a control flow edge to exception handlers.

3.1.2 Determining Control Flow Edges

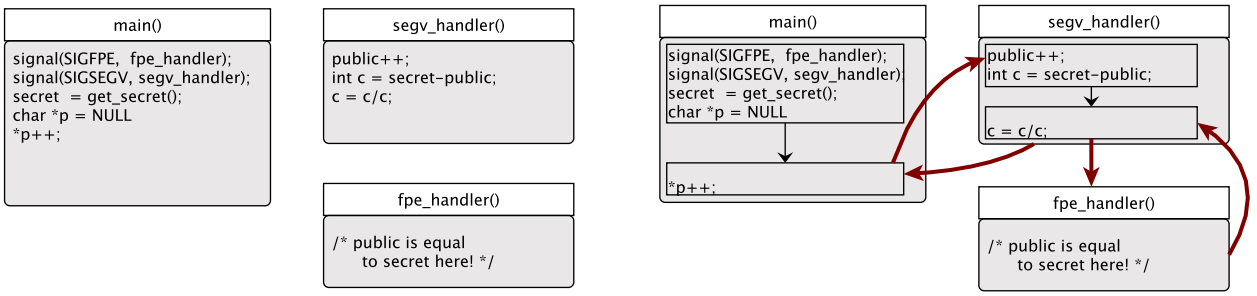
As noted earlier, a challenge in identifying the possible control transfer edges from instructions to their intended exception or signal handlers is that this requires specific run-time information that is not available statically. For synchronous signals, we use symbolic execution to determine the possibility of an exception during the course of an instruction's execution. This is done by constructing the path constraint along the execution path and sending appropriate queries to the SMT solver on instructions that can potentially raise exceptions. The idea is similar to those using symbolic execution for automatic fault detection [9] and exploit generation [7, 14]. The following example describes the idea more clearly.

EXAMPLE 3.1. Consider the following code fragment:

```
1   r0 := input()
2   r1 := r0 − 10
3   r2 := r0 − 20
4   r3 := r1/r1
5   if r0 > 20:
6       r3 := r2/r2
```

Instruction 1 gets an input value and copies it to the register r_0 and instructions 2 and 3 calculate $r_0 - 10$ and $r_0 - 20$ in r_1 and r_2 respectively. Instruction 4 performs a division operation which can raise an exception if the divisor (r_1) is zero. Knowing the symbolic expression for r_1, expression $r_1 == 0$ is satisfiable for inputs equal to 10, instruction 4 can raise an exception and, if there is a handler registered for divide-by-zero exception, a control flow edge should be added from instruction 4 to the the handler function. Similarly for instruction 6, inputs of 20 can cause an exception to be raised but since the path constraint mandates $r_0 > 20$ and so $r_2 > 0$, the formula $r_2 == 0$ is not satisfiable and the instruction is immune to divide-by-zero exceptions. ∎

In the symbolic execution engine we have implemented three major synchronous exceptions that are commonly used



(*a*) Static CFG

(*b*) Static CFG augmented with control flow edges to signal handlers

Figure 2: Static and augmented static CFGs for the code in Figure 1(a)

to obfuscate normal control flow in malicious codes and/or in binary packer tools [21]. Our symbolic execution engine constructs appropriate constraints for synchronous exceptions that have a user exception handler registered for it. As mentioned earlier, we are interested in finding exceptions that are triggered based on some tainted or symbolic condition or value. For this purpose, the symbolic execution first checks whether a particular instruction has the desired characteristics, and then builds a formula to represent the exception and sends it to the SMT solver to check its satisfiability. A few of the most important exceptions and how they are handled by the symbolic execution engine are as follows:

- `SIGSEGV` : This exception is caused by memory access violations such as accessing a memory location that is not available to the process. For memory pointers that are tainted, the symbolic execution engine constructs a constraint that checks whether the symbolic pointer can point anywhere that is not a legal memory address at that point in the program. If the constraint is satisfiable, the constraint solver finds an input that will cause the pointer to point to an illegal address that if fed into the program and transfers control to the signal handler for `SIGSEGV`

- `SIGFPE` : This exception is raised in case of arithmetic errors such as divide by zero. In case of a divide instruction, the symbolic execution engine constructs a constraint that checks whether the dividend used in the instruction can be zero based on some inputs, if the dividend is symbolic. If such an input exists, it will report it to the user.

- `SIGILL` : This exception is raised when an instruction opcode is not recognizable by the `CPU`. If in the execution of the program, an instruction is being overwritten with a symbolic value [56], the symbolic execution engine constructs a constraint to check whether the opcode that is being written can be illegal based on some input.

Using symbolic execution to identify exception-based control transfers has two advantages. First, the analysis can find alternative inputs to trigger alternative execution paths in the code other than those paths existed because of normal control flow structure such as condition jumps. One important application of this is in the analysis of malicious or obfuscated code that hide their control flow through exception-based control transfers. Secondly, the alternative paths that are found can be used towards implicit information flow detection that otherwise was not possible to detect because of missing control transfer edges in the static CFG.

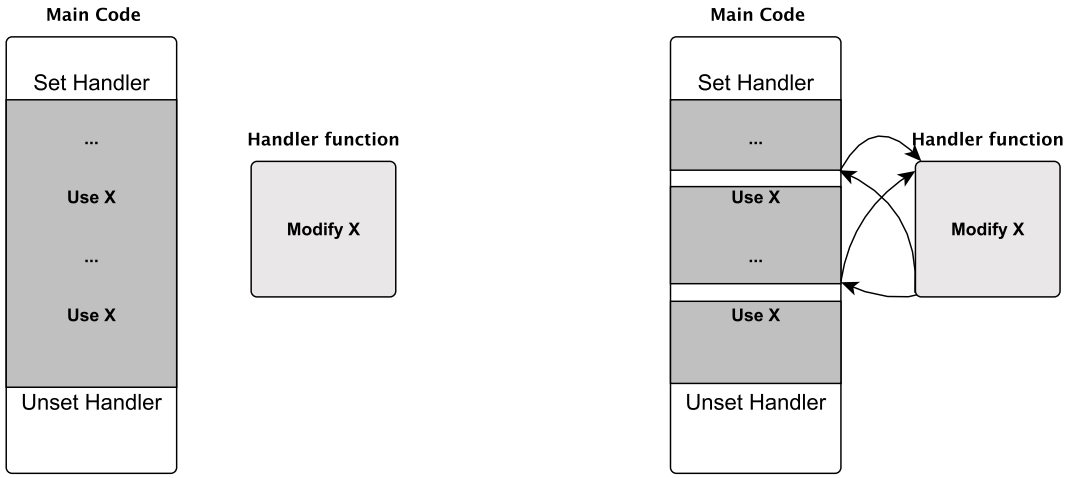
3.2 Asynchronous Events

Asynchronous signals are those that are not a direct result of the execution of an instruction, but rather are generated by some external sources and delivered to the process through the operating system (e.g. Figure 1(b)). Asynchronous signals are more challenging than synchronous ones to handle and can cause inaccuracies in both symbolic execution and dynamic taint analysis because the time an exception handled by the program receving it may change the execution path and/or the data flow equations.

Similar to CFG augmentation for synchronous exceptions discussed in Section 3.1, we use the augmentation technique for unexpected control transfers caused by asynchronous signals. The only difference here is that since there is no fixed point in the code where the control diverges from the main execution thread, we need to analyze different possibilities and account for different situations. Of course the first non-trivial solution is to add a new control flow edge following every instruction in the main execution trace where there is a possibility of control divergence, but this imposes unnecessary computational complexity to the analysis.

To handle asynchronous signals, in general, we need to take into account the side effects of executing the handler code. Figure 3 tries to visualize this situation where as opposed to synchronous signals in which the control flow edge is at a fixed location (in the code), for asynchronous signals, control transfer points are determined by the possible locations that might be impacted by executing the handler function. As shown in Figure 3(a), variable `X` is defined globally where it can be accessed from both the main code and the signal handler. Suppose that the gray box in the main code is where a signal can be delivered to the process meaning that the control could be transferred to the signal handler at any point in the gray area. The variable `X` is accessed multiple times in the gray area, while the signal handler function modifies the variable `X`. Modification to `X` may have impact on the main code where `X` is used, i.e. points where there is an edge to the handler in Figure 3(b). Figure 3(b) shows the augmented CFG for the code and handler of Figure 3(a).

To handle asynchronous signals we are not necessarily proposing to use symbolic execution to identify alternative



(*a*) Static CFG of the code. The grey box in the main code appears as a single basic block

(*b*) Augmented CFG for asynchronous signal handler

Figure 3: CFG Augmentation for asynchronous signal handler

execution paths caused by signals. Although symbolic execution can be helpful in some cases, such as when the argument to the `kill` function (that is used to send arbitrary signals to processes) is symbolic. This results in activation of different functions in the receiving process when there are different handlers registered for different signals, and symbolic execution can be used to help identifying those inputs that lead to different handlers being called in the execution. As mentioned earlier, asynchronous signals are generally generated externally which are then delivered to the process. However for an attacker to exploit these mechanisms provided by the OS, they need to arrange for the signals to be delivered to the process somehow, perhaps by forking multiple threads and have the threads to communicate with each other with signals and this arrangement that is necessary for an attack to succeed, makes the attack detectable and perhaps predictable.

The code example that are shown in this section for use `SIGALRM` signal as to show how to use asynchronous signals for control transfer purposes, however the idea that is discussed as a proposed solution is generalizable to the other types of asynchronous signals. Examples of other signals are those that are used for inter-process communications such as `SIGCHILD` that is raised when a forked child of a process dies or `SIGUSR1` that is left for user defined handlers.

3.3 Attack Model

As with other techniques that rely on symbolic execution and SMT solvers, our approach is limited by the theoretical boundaries of SMT solvers [48, 53]. An attacker can also add complexity to the code in order to add complexity to the path constraints. This can increase the processing time for the back-end SMT solver to check satisfiability of the path condition. A related attack is to use code that produces too much work for the analysis. For example by using opaque symbolic memory pointers (symbolic pointers that are not unsatisfiable), the analysis needs to send a lot of queries to the SMT solver which slows down the analysis [56]. In order to reduce the effects of these types of attacks, we have incorporated a chaching mechanism built in to our prototype implementation that reduces the number of queries sent to the solver significantly, limitting the possibility of these types of attacks (see Sections and). However, note that in the context of identifying the possible exception-based control-flow behaviors of a program, the back-end SMT solver is used only for those exceptions for which exception handlers have been registered during execution and for those instructions that can give rise to such exceptions, not for all executed instructions.

4. IMPLEMENTATION

We have implemented a prototype system based on Intel PIN tool [34] which is a dynamic binary instrumentation tool. Our system works on the binary level and so we do not need access to the source code. The analysis is based on two main components: 1) dynamic taint analysis; and 2) symbolic execution. Our dynamic taint analysis engine takes the approach used in [55]. Our taint analysis uses bit-level granularity for taint taint propagation by using generic taint labels for inputs to the program. Our symbolic execution engine is a layer on top of the taint analysis engine which interfaces with STP [23] SMT library to solve symbolic expressions. We have implemented concolic testing that runs the input program with the given inputs and analyses the execution trace that is observed along the execution path. Both taint analysis and symbolic execution analyses are done on the `X86` instruction set.

For dynamic taint analysis, we use `objdump` tool on Linux to produce disassembly and create CFG of the disassembled code. The post-dominator information then is extracted by analyzing the CFG which specifies post-dominators for conditional branches in the code. For signal handlers, however, we use PIN APIs to intercept user's registered signal handlers to analyze them. Our current implementation does not analyze the handler code statically because static analysis of binary code is hard in general and does not always guarantee cor-

rect results [33] due to code obfuscation or self-modification that are common among malware code. We instead take a dynamic approach where the handlers are intercepted at run-time and capture the implicit flow by marking those data flows that are tainted because of the exception-based control transfers. We realize that with this approach it is possible to miss some opportunities for symbolic execution, but the alternative assumption that we make here is quite realistic as we have observed in our evaluations with malicious codes.

To find specific user handlers that are registered by the user code, the tool intercepts the calls to `sigaction` and `signal` system calls that are used to register user handlers for given signals. Using the addresses of the signal handlers that are registered by the user, we are able to augment the static CFG and add explicit control flow edges whereever the symbolic execution engine finds a satisfiable input that would cause an instruction to raise a particular exception.

4.1 Symbolic Pointer Caching

A naive implementation of the ideas described above would query the SMT solver at every runtime memory reference. Since programs can have a large number of memory references, and the cost of querying an SMT solver can be quite large, this would result in unacceptably high runtime overheads. To mitigate this problem, we use a symbolic pointer caching mechanism that reuses the results of prior queries to the SMT solver where this is guaranteed to be sound, thereby reducing the number of queries to the solver.

In the x86 architecture, the most general form of an address computation involves two integer constants and two registers (the base and index registers) as follows:[1]

$$address = baseReg + scale \times indexReg + displacement.$$

We refer to this as an *address expression*. A pointer is marked as symbolic if, in its address expression, either the base register or index register is symbolic; the symbolic term of such a pointer is *baseReg* if the base register is symbolic and $scale \times indexReg$ if the index register is symbolic. All other terms can be considered as constant terms. If such a pointer is used to trigger a SIGSEGV, then at least one of the possible addresses that can be generated by the pointer must illegally reference a page in memory. Thus, by checking to see if a symbolic pointer can access an invalid page in memory it can be determined if the program will conditionally trigger a SIGSEGV. By taking advantage of the fact that an entire page will be marked as invalid, not just a particular memory location, we can cache and reuse results from previous queries sent to the solver.

Consider two pointers A and B. If the address expressions for A and B have the exact same symbolic and constant terms, then obviously they will access memory the exact same way. Thus we know that if it is possible for pointer A to illegally access a page in memory, then it must be possible for B to make an illegal access as well. Now let pointers A and B share the same symbolic terms, but not the same constant terms. Since the symbolic terms are the same, we know that for every memory location A can access, there must exist a corresponding memory location that B can access at some displacement d away, for some constant d. It is therefore possible that the displacement by d can cause B to access pages that A did not, meaning that we can no longer reason about B from A. If, however, d is less than the size of a memory page (typically 4096 bytes) then B's corresponding access must either fall on the same page as A's, or on one of the adjacent pages. As a result, a third pointer C, which has the same symbolic terms as A and B but a displacement which places it between A and B, must only be able to access pages in memory that could be accessed by A or B. To that end, if both of them can only access memory pages legally, then C can only access pages legally. Likewise if both A and B can access a page illegally, then C must also be able illegally access a page. However, C cannot be reasoned about if A and B disagree since there would be an unknown boundary between A and B that determines C's state based on where it lies in relation to that boundary. With this information, a caching mechanism can be constructed. Whenever a symbolic pointer must be checked, we can construct and check a second symbolic pointer with the same symbolic terms and an offset less than a page. If the decisions from the solver for these two pointers agree, we can save the result for future reuse.

The cache has two parts: a storage mechanism capable of holding pointers whose solver decisions can be reused and; an invalidation mechanism to discard pointers whose symbolic terms have been changed. For the storage portion, we use a structure that maps a symbolic value to a list of entries whose pointers use that value. If the entries corresponding to a particular symbolic value need to be invalidated, the list corresponding to the value can simply be cleared. This means that to find a symbolic pointer, we have to check that all symbolic values used by the pointer can locate an entry in the cache. If an entry is not found, then that symbolic value must have been invalidated and any entries found are dirty. We use this structure because it makes invalidation, which occurs more frequently than searches, fast and easy.

To maintain the pointer validity requirements placed upon the cache, a strict set of rules is used to determine when to remove a pointer from the cache. If a symbolic pointer has an entry in the cache, the symbolic values it uses can only be modified by additions and subtractions of constants with no other symbolic operands. This guarantees that the symbolic terms of the pointer are the same when a cache entry is reused. If a symbolic value with entries in the cache is modified in any other way, all entries that use the value are removed from the cache. Additionally, if the program modifies the protection of any page in memory, the entire cache is flushed since it is not known which, if any, of the cache entries are affected.

To improve the performance of the cache, a few additional features are built in as well. The first is the ability to track symbolic operands if they are moved. Programs often temporarily store the operands needed for a pointer in registers, then once they have completed their memory access will either replace them or move them back to memory. We track the movement of the symbolic values to ensure that we can recognize when two symbolic pointers are the same without having to perform an expensive comparison between the constraints. A second optimization involves symbolic pointers in loops. Very often, the same symbolic pointer is used within a loop whose termination condition is based on a symbolic value. Since each iteration of the loop changes the path constraint to code in the loop body, a straightforward approach to cache management would invalidate cache entries that use

[1]The x86 architecture supports a number of other addressing modes that can be seen as special cases of that described here.

the symbolic value, since any change to the path constraint potentially implies a change to the symbolic value as well. This would cause a large number of cache misses within loop. However, a conditional can only further constrain a symbolic value and therefore a symbolic pointer. Thus, if a pointer can only make legal memory accesses, another pointer that can only access a subset of the same memory locations can also only access legal locations. If, on the other hand, a pointer can access an illegal location, it is possible that after being constrained the same pointer can only access legal locations. Therefore, the cache is only used for symbolic pointers that can only access valid memory regions and therefore cannot cause a SIGSEGV. Since the number of symbolic pointers that can access invalid memory regions are in the minority and many of them would be invalidated by a conditional anyway, this decision does not have a large impact on the hit rate of the cache.

5. EXPERIMENTAL EVALUATIONS

We evaluated our prototype tool against a variety of different test programs and malicious codes and we compared our results with current state-of-the-art available tools in both dynamic taint analysis and symbolic execution. The samples were all run on a Linux virtual machine running Ubuntu 14.04 which was given 8 processors and 8 GB of ram.

5.1 Dynamic Taint Analysis

For dynamic taint analysis we used two sets of programs for the evaluations:

- The first set consists of three sample test programs that use exception-based control transfers to obfuscate the branch points in the program. All three programs define exception handlers that will redirect the control, and have some input value that is used to conditionally trigger the exception. Additionally, all three samples propagate some tainted data in the handlers to resemble the idea of implicit information flow. These three programs are:

  1. *invalid-memory* uses user input to create a memory pointer that is a valid memory address if the input satisfies some conditions, otherwise an invalid pointer.
  2. *invalid-opcode* overwrites `nop` instructions in the code by an invalid opcode if the user input does not satisfy the conditions resulting in a `SIGILL` in the code.
  3. *divide-by-0* performs a division where the dividend is computed from input. The dividend is zero for inputs not satisfying the condition and non-zero otherwise.

- The second set consists of two exploit codes written in `C` that have behaviors similar to what is described in this paper. These are proof-of-concept exploits that are written for different vulnerabilities and are publicly available through web-sites such as http://www.exploit-db.com. The samples overwrite signal handlers for particular signals. The exploits are mentioned by their corresponding CVEs:

  1. *CVE-2004-1235* exploits a race condition in `load_elf_library` and `binfmt_aout` function calls that exists in some Linux kernel versions, allowing an attacker to execute arbitrary code by manipulating the VMA descriptor. The exploit sets handler for `SIGALRM` signal and uses it to redirect control flow to propagate data.
  2. *CVE-2005-0736* exploits an integer overflow in `sys_epoll_wait` for some Linux kernels allowing an attacker to overwrite kernel memory with a large number of events. This sample also sets exception handlers for different signals that guide the execution in case of an unexpected event.

We found Dytan [17] the only dynamic taint analysis system that was available publicly and implements implicit taint propagation through control flow edges. For the test-cases that contained some form of implicit taint propagation through the signal handlers, we ran experiments with Dytan and our tool and the results are presented in Table 1. Dytan failed on all the test-cases while our tool was able to successfully identify the implicit flows caused by signal handlers, while the second exploit uses the handler to handle possible memory faults by setting variables that will eventually change the control flow of the code executed afterwards. We have identified these instances as implicit flow since the control flow caused by exceptions are implicitly used to set variables that consequently affect the flow of execution.

	Dynamic Taint Analysis	
Programs	Dytan	Our System
invalid-memory	×	✓
invalid-opcode	×	✓
divide-by-0	×	✓
CVE-2004-1235	×	✓
CVE-2005-0736	×	✓

Table 1: Dynamic taint analysis results

5.2 Symbolic Execution

We evaluated our prototype against different malicious programs procured from `vxheaven`[2] and compared our results against current state-of-the-art symbolic execution tools.

To evaluate the symbolic execution engines, we started with a base set of the following Linux malware:

1. *Caline* is a simple linux virus that infects ELF files resulting in a simple message being placed in the binary.
2. *w00lien* that has various malicious capabilities, such as spawning a shell for remote connections, self-encryption and file infection.
3. *lacrimae* is a malware mutation engine which reads in an ELF file and writes back a mutated version of the binary.
4. *rapeme* is a RPM archive infector virus. It finds and infects an RPM file with a malicious payload and then recomposes the infected code into an RPM.
5. *kaiowas11* is a proof-of-concept showing run-time binary encryption/decryption.

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

For each of the malware sources, we selected a conditional or a set of conditionals in the malicious program that an analysis tool needed to explore in order to discover all of the behavior the malware could exhibit. The selected statements were typically conditionals important to the integrity of the malicious algorithm, or used for anti-analysis. For each sample, we then created two more samples by transforming the conditionals into an equivalent statement that instead branched using conditional exceptions by moving the code for true branch into the signal handler function registered for that particular signal. The first obfuscated sample uses an asynchronous conditional exception using `alarm` signal (similar to Example 1(b)) while the second sample uses illegal instruction exception (`SIGILL`). For samples using illegal instruction, the conditional code is transformed to a call to a buffer that contains illegal instructions, however the buffer is overwritten by `nop`s if the condition is evaluated to false. This results in the execution to continue the false branch in case the condition is not met, otherwise the control is transformed to the exception handler that contains the code for true branch.

In order to evaluate our ideas against state-of-the-art symbolic execution engines, we picked a handful of available symbolic execution engines and compared their results with ours. The symbolic execution engines that we used are *KLEE* [7], *S2E* [16], *FuzzBall* [4] and *angr* [49]. KLEE and S2E are automated test-case generation symbolic execution engines that maximize the code coverage, and FuzzBall and angr are symbolic execution frameworks. KLEE needs to have access to source code while the others work at the binary level. Test-cases were annotated to introduce symbolic variables for KLEE and S2E. FuzzBall and angr however accepts arguments which instructs the tool to mark inputs, memory locations or registers as symbolic at certain point in the execution.

The results are summarized in table 2. All of the tools were able to discover all of the targeted paths in the original unmodified programs. The number of paths (based on the conditional statements that we targeted for our obfuscation) is reported under the second column. The only error that we observed was that in the rapeme test, angr discovered 4 additional paths that were technically impossible to execute. Once the conditional exceptions were inserted, however, the state-of-the-art competition had difficulty discovering the alternative paths. The next 5 columns show the number of paths discovered by the tools in the test programs obfuscated using `ALARM` signal, and the remaining 5 columns show the same results for the test programs obfuscated using illegal instruction.

Only one of the state-of-the-art competition tools was able to successfully handle the asynchronous alarm signal. Neither angr nor KLEE supported calls to `alarm` with symbolic argument, so to allow these two engines to progress past the alarm call we had to hook the calls to alarm (and `sleep` for FuzzBall) and return the appropriate value. This allows them to explore the false path, but since neither of them know anything about the semantics of the alarm call in this case, they cannot properly explore all guarded paths. KLEE, on the other hand, has 3 different versions of `libc` that can be used to emulate the library functions. We tried running the samples on all versions of `libc` available, but none of them handled the alarm call so that the true path would be discovered. S2E was the only engine that handled the asynchronous exception guard correctly. We believe that this is because S2E has access to code in the kernel and in user mode. With this, we believe it was able to observe how the alarm signal is triggered by the kernel and thereby could produce the correct results.

None of the other tools were able to handle the synchronous illegal instruction exception. KLEE only works on the C source code, and so it is unable to handle programs that modify their own binary like we did in this example. The other tools, on the other hand, do have the ability to handle self-modifying code. None of them, however, knew how to handle an illegal instruction. Rather, they all reported an error and exited the analysis. To overcome this, we had to use facilities built into the engines to `nop` an illegal instruction if one is encountered. This allowed execution to continue past the illegal instruction and always discover one path of the conditional statement. None of them, however, were about to reason about other instructions that could have been written into the buffer to cause alternative behavior. Our tool handles illegal instructions by keeping track of the instruction bytes that are modified using a symbolic value. When encountered with an illegal instruction, our tool constructs a symbolic expression involving the path constraint and asks the SMT solver if the expression can be solved so that the instruction can be a `nop`. If solvable, the results from the SMT solver can be used to determine the necessary input that could cause the alternative path to be taken.

5.3 Performance

5.3.1 Overhead

Table 3 shows the running time of each obfuscated sample versus the analysis time of our system and FuzzBall. All the times are in seconds and for our system, the analysis time includes both dynamic taint analysis and symbolic execution. The samples on the row numbered with 1 are obfuscated using `alarm` while the samples on row labeled with 2 are obfuscated using illegal instruction. To do a fair comparison, we have only compared our system to FuzzBall since it's behaviour is closest to that of our system. FuzzBall automatically discovers alternative execution paths if it finds any symbolic conditional statements in the code, but since it does not handle the obfuscated conditionals in our samples, it only analyzes the execution path based on the given input. This is similar to the behaviour of our tool since our tool does not automatically execute the alternative paths but reports any possible input value that would trigger them. As it can be seen from the table, the performance of our tool is generally better than FuzzBall and the overall analysis time remains in reasonable ranges.

5.3.2 Caching Improvements

To measure the efficacy of caching, we ran our tool on a modified version of *Md5* from http://people.csail.mit.edu/rivest/Md5.c since it is both computation and memory intensive. We chose *Md5* simply because our test cases do not cause a lot of interactions with the SMT solver. The original program was modified with a simple `SIGSEGV` handler and calls to force our prototype to generate many tainted pointers for some of the operations. We ran the program on three different input files as shown in Table 4. The first column in Table 4 shows the `CPU` time required to process the input file without analysis, while the second column shows the analysis

Programs	# of paths in original code	Number of discovered paths in obfuscated samples									
		Obfuscated using `alarm`					Obfuscated using illegal instruction				
		KLEE	FuzzBall	angr	S2E	Our System	KLEE	FuzzBall	angr	S2E	Our System
Caline	2	1	1	1	1	2	N/A	1	1	1	2
w00lien	2	1	1	1	2	2	N/A	1	1	1	2
lacrimae	2	1	1	1	2	2	N/A	1	1	1	2
rapeme	4	1	1	1	4	4	N/A	1	1	1	4
kaiowas11	2	1	1	1	2	2	N/A	1	1	1	2

Table 2: Symbolic execution results

Workload	Normal Exec. Time (s)	Analysis Exec. Time (s)	Total Query Time (s)	No. Queries	Cache Hits	Cache Misses
4GB	0.065	157.633	3.346	268562	268496	66
1.5GB	0.023	54.420	2.808	90955	90889	66
75MB	0.004	6.624	2.646	4661	4595	66

Table 4: Caching performance of our prototype tool for different workloads on MD5

	Programs	Execution times (s)		
		Original	FuzzBall	Our System
1	*Caline*	1.002	2.522	1.838
	w00lien	1.166	2.539	2.019
	lacrimae	1.000	2.588	2.174
	rapeme	2.048	18.763	4.909
	kaiowas11	1.001	5.392	1.791
2	*Caline*	0.002	4.151	1.014
	w00lien	0.093	4.336	0.831
	lacrimae	0.002	4.551	0.638
	rapeme	0.046	71.315	2.723
	kaiowas11	0.001	18.173	0.796

Table 3: Cost analysis of our prototype tool

time. Third column shows the time spent in the solver in seconds, and the rest of the columns show the number of queries that our tool encountered during the analysis, number of queries that were found in cache and the number of queries which needed to be sent to the solver. Under these workloads, cache hit rate is generally above 98% suggesting that the caching mechanism provides a lot of benefit when a pointer repeatedly accesses the same page(s) (e.g. in a loop). As the number of hits decreases, it can be seen that the time spent in the solver increases, so if there are a lot of misses in the cache, the solver will require a large portion of the runtime.

6. RELATED WORK

Many researchers have investigated symbolic code execution; see the survey by Schwartz *et al.* [45]. An important application of this approach is in analysis of malicious and/or obfuscated code [1,3,6,18,36,47,58,59]. However, these works generally do not explicitly address the challenges that arise in analyzing obfuscated code, which is especially prevalent in malware. Yadegari and Debray discuss approaches to symbolic analysis of obfuscated code [56], but this does not consider exception-based control transfers.

Our approach to find exception-based control trnasfers are similar to those used in existing systems such as KLEE, S2E or Mayhem. These systems use symbolic execution to guide the execution of the program under the analysis and find different combination of inputs that cause the program to crash [7,9] which then can be combined with other reasonings to automatically generate exploits [14]. This is different from our work that recognizes program faults as an obfuscation technique to obscure control flow transfers. [19] discusses the execution of enabled interrupts in analyzing firmware code but their approach is limitted to small binaries that are written in C and thus require source code. Their approach is not applicable to native large applications and easily results in state explosion.

The use of exceptions to obfuscate control flow is well known [40]. Malware have long used a simple instance of this approach to obfuscate direct unconditional jumps, by constructing and dereferencng a null pointer. In commonly encountered malware, this obfuscation is typically used to bypass ordinary anti-virus detectors rather than to propagate information through implicit flows. However, it is straightforward to modify this code to use exception-based control transfers to hinder dynamic taint propagation and symbolic execution.

A number of researchers have described security-related applications of dynamic taint analysis [26,27,29,38,54]. Clause *et al* [17], Schwartz *et al.* [45], Song *et al.* [52] and Nethercote *et al.* [37] discuss general frameworks for dynamic taint analysis, but do not address issues arising from implicit flows in obfuscated code, and exception-based control transfers in particular. The problems arising from dynamic taint analysis of code containing implicit information flows is discussed by Cavallarro *et al.* [12].

A number of researchers have looked into the problem of analysis of exception-handling behavior of programs. This work typically focuses on explicit exception-management mechanisms (throw-catch statements) at the source code level [5,15,28,43,50,51]. We are not aware of any work on reasoning about exception behavior at the binary level.

7. CONCLUSIONS

While dynamic taint analysis and symbolic execution have

a number of important applications in security-related program analyses, existing techniques for these analyses have trouble dealing with many of the code obfuscations employed by malicious programs. A particular example of this is the use of exceptions to obfuscate control transfers. This paper discusses the problems that dynamic taint analysis and symbolic execution systems can encounter when analyzing programs containing implicit information flows arising from exception-based control transfers. We propose a generic solution for code where such exceptions are handled via user-defined exception handlers. Experimental results using a prototype implementation show that our approach yields better results than existing analysis techniques.

Acknowledgments

This research was supported in part by the National Science Foundation (NSF) under grants III-1318343, CNS-1318955, and CNS-1525820.

8. REFERENCES

[1] U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. 16th Annual Network and Distributed System Security Symposium (NDSS 2009)*, Feb. 2009.

[2] U. Bayer, P. Milani Comparetti, C. Hlauscheck, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS)*, Feb. 2009.

[3] U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1), Aug. 2006.

[4] Bitblaze. FuzzBALL: Vine-based Binary Symbolic Execution.

[5] M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 1–12. ACM, 2009.

[6] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection: Countering the Largest Security Threat*, volume 36, pages 65–88, 2008.

[7] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[8] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *Model Checking Software*, pages 2–23. Springer, 2005.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[10] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071. ACM, 2011.

[11] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2), 2013.

[12] L. Cavallaro, P. Saxena, and R. Sekar. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Stony Brook University, Stony Brook, New York*, 2007.

[13] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.

[14] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy*. IEEE, 2012.

[15] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural exception analysis for java. In *ACM Symposium on Applied Computing*, 2001.

[16] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Mar. 2011.

[17] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 196–206. ACM, 2007.

[18] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: detecting hidden malware timebombs with virtual machines. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 25–36, Oct. 2006.

[19] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, 2013.

[20] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[21] P. Ferrie. Anti-unpacker tricks–part one. *Virus Bulletin*, 4, 2008.

[22] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification*. Springer, 2007.

[23] V. Ganesh and T. Hansen. STP. `https://github.com/stp/stp`.

[24] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2005.

[25] W. G. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 175–185. ACM, 2006.

[26] C. Hauser, F. Tronel, L. Mé, and C. J. Fidge. Intrusion detection in distributed systems, an approach based on taint marking. In *Proc. 2013 IEEE International Conference on Communications (ICC)*, pages 1962–1967, 2013.

[27] C. Hauser, F. Tronel, J. F. Reid, and C. J. Fidge. A taint marking approach to confidentiality violation detection. In *Proc. 10th Australasian Information Security Conference (AISC 2012)*, Jan. 30 2012.

[28] J.-W. Jo and B.-M. Chang. Constructing control flow graph for java by decoupling exception flow from normal flow. In *Computational Science and Its Applications–ICCSA 2004*, pages 106–113. Springer, 2004.

[29] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[30] J. Kong, C. C. Zou, and H. Zhou. Improving software security via runtime instruction-level taint checking. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 18–24. ACM, 2006.

[31] H. Lin, X. Zhang, M. Yong, and B. Wang. Branch obfuscation using binary code side effects. In *International Conference on Computer, Networks and Communication Engineering (ICCNCE 2013)*. Atlantis Press, 2013.

[32] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection*, pages 338–357. Springer, 2011.

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

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[35] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[36] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[37] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 89–100, June 2007.

[38] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[39] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Recent Advances in Intrusion Detection*, pages 124–145. Springer, 2005.

[40] I. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Proc. Usenix Security 2007*, pages 275–290, Aug. 2007.

[41] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 135–148. IEEE, 2006.

[42] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su. A framework for understanding dynamic anti-analysis defenses. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 2. ACM, 2014.

[43] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):191–221, 2003.

[44] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *SECRYPT*, pages 461–468, 2013.

[45] E. J. Schwartz, T. Avgerinos, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symposium on Security and Privacy*, pages 317–331, 2010.

[46] K. Sen, D. Marinov, and G. Agha. 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*, pages 263–272, Sept. 2005.

[47] M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic reverse engineering of malware emulators. In *Proc. 2009 IEEE Symposium on Security and Privacy*, May 2009.

[48] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proc. 15th Network and Distributed System Security Symposium (NDSS)*, Feb. 2008.

[49] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[50] S. Sinha and M. J. Harrold. Analysis of programs with exception-handling constructs. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 348–357. IEEE, 1998.

[51] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[52] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proc. of the 4th International Conference on Information Systems Security*, Dec. 2008.

[53] Z. Wang, J. Ming, C. Jia, and D. Gao. Linear obfuscation to combat symbolic execution. In *Computer Security–ESORICS 2011*, pages 210–226. Springer, 2011.

[54] W. Xu, S. Bhatkar, and R. Sekar. Practical dynamic taint analysis for countering input validation attacks on web applications. Technical report, Technical Report SECLAB-05-04, Department of Computer Science, Stony Brook University, 2005.

[55] B. Yadegari and S. Debray. Bit-level taint analysis. In *International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014.

[56] B. Yadegari and S. Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, ser. CCS*, volume 15, pages 732–744, 2015.

[57] X. Yao, J. Pang, Y. Zhang, Y. Yu, and J. Lu. A method and implementation of control flow obfuscation using SEH. In *Multimedia Information Networking and Security (MINES)*, pages 336–339. IEEE, 2012.

[58] H. Yin and D. Song. Analysis of trigger conditions and hidden behaviors. In *Automatic Malware Analysis*, SpringerBriefs in Computer Science, pages 59–67. 2013.

[59] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, 2007.