1

Automatically Identifying Trigger-based Behavior in Malware

David Brumley, Cody Hartwig, Zhenkai Liang,
James Newsome, Dawn Song, Heng Yin
Carnegie Mellon University
{dbrumley,chartwig,zliang,jnewsome,dawnsong,hyin}@cmu.edu

Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Malware often contains hidden behavior which is only activated when properly triggered. Well known examples include: the MyDoom worm which DDoS's on particular dates, keyloggers which only log keystrokes for particular sites, and DDoS zombies which are only activated when given the proper command. We call such behavior *trigger-based behavior*.

Currently, trigger-based behavior analysis is often performed in a tedious, manual fashion. Providing even a small amount of assistance would greatly assist and speed-up the analysis. In this chapter, we propose that automatic analysis of trigger-based behavior in malware is possible. In particular, we design an approach for automatic trigger-based behavior detection and analysis using dynamic binary instrumentation and mixed concrete and symbolic execution. Our approach shows that in many cases we can: (1) detect the existence of trigger-based behavior, (2) find the conditions that trigger such hidden behavior, and (3) find inputs that satisfy those conditions, allowing us to observe the triggered malicious behavior in a controlled environment. We have implemented MineSweeper, a system utilizing this approach. In our experiments, MineSweeper has successfully identified trigger-based behavior in real-world malware. Although there are many challenges presented by automatic trigger-based behavior detection, MineSweeper shows us that such automatic analysis is possible and encourages future work in this area.

1.1 Introduction

In many malware programs, certain code paths implementing malicious behaviors will only be executed when certain *trigger conditions* are met [15, 18, 23, 24]. We call such behavior *trigger-based behavior*. Trigger-based behavior may be set off by many different *trigger types*, such as time, system events, and network inputs.

For example, many viruses attack their host systems on specific dates, such as Friday the 13th or April Fool's Day [18, 24]; worms may launch attacks at specific times [13], some keyloggers only record keystrokes to files when the application window name contains certain keywords [15]; some browser-helper-object-based spyware only logs information if the URL contains a certain keyword [23]; some distributed denial-of-service tools only start launching attacks when receiving certain network commands [3]. Thus, trigger-based behavior is a real problem, causing millions of dollars of damage [15, 18, 23–27], and detecting trigger-based behavior is important for understanding the malware's malicious behavior and for effective malware defense.

Currently, trigger-based behavior is often analyzed in a tedious, manual process. To the best of our knowledge, there is no previous work on automating trigger-based behavior analysis. Given a piece of potentially malicious code, a typical manual analysis scenario is as follows: a) the analyst runs the malware in a virtual machine and may observe nothing since the trigger condition may not be met, b) he may then perform some disassembly and build up a mental model of the program execution, c) he may then guess which parts of the input or system setup to change and rerun the malware and hope to observe something new. This process is repeated until the analyst runs out of time, patience, or gets lucky and uncovers the trigger-based behavior. Such a manual process is slow, labor intensive, and does not scale.

These problems apply directly to botnets. From an analyst's point of view, a bot is a malicious binary containing many hidden behaviors. Using the framework we describe an analyst can find the behavior a certain bot exhibits including actions it takes and commands it responds to. We have specifically researched this application in our most recent work [4].

Our Approach. In this chapter, we propose that *automatically* identifying and reasoning about trigger-based behavior in malware is possible, and design a system as a first step towards this goal. In particular, we show how to design and integrate techniques from formal verification, symbolic execution, binary analysis, and whole-system emulation and dynamic instrumentation to enable automatic identification and analysis of trigger-based behaviors in malware. Automatic trigger-based behavior detection is an extremely challenging task. For example, completely automatic analysis of trigger-based behavior for all programs is undecidable (Section 1.5). However, we show that our approach can provide great value in many cases. Our system, *MineSweeper,* is able to automatically identify the trigger-based behaviors in several real-world malware examples. Even when complete automatic analysis is not possible, we design our system so that it still provides valuable information about potential trigger-based code paths which a human would otherwise have to discover manually.

To design an approach for automatic trigger-based behavior analysis, we first observe that at a high level, triggers in a program are implemented as conditional jumps depending on inputs from the trigger types of interest such as time, keyboard, or network inputs. The malicious code is triggered when the conditional jumps evaluate to the desired directions, e.g., the current time is equal to the trigger time. There-

fore, given trigger types of interest, one key to uncovering trigger-based behavior is to construct values for trigger inputs (i.e., inputs from trigger types of interest) that makes the conditional jumps evaluate in the desired direction, activating the trigger-dependent code. We call the condition that the trigger inputs need to satisfy in order for the code execution to go down a path uncovering the trigger-based behavior the *trigger condition*, and the values of the trigger inputs satisfying the trigger condition the *trigger values*. Second, we observe that trigger-based behavior could be embedded at any point in the program. Thus, we need to be able to explore many different program paths which could depend on trigger inputs.

From these observations, we design an approach as a first step towards automatic trigger-based behavior analysis in malware. Our approach takes as inputs the binary program of the malware to be analyzed and a set of trigger types. In order to automatically explore trigger-based behavior in the program based on the given trigger types, we employ *mixed concrete and symbolic execution* to automatically and iteratively explore different code paths which could depend on trigger inputs. In particular, trigger inputs are represented symbolically, and instructions that depend upon the trigger inputs operate on symbolic values, and are executed symbolically. Conversely, instructions that do not depend on trigger inputs operate on concrete values, and are concretely (natively) evaluated (for efficiency). Thus, symbolic execution builds up symbolic formulas over the symbolic inputs (which are in turn based on the trigger types). Note that the ability to mix concrete and symbolic execution is important to reduce the formula size. As our experiments indicate, almost all instructions can be concretely executed.

For any path to be explored, the mixed concrete and symbolic execution automatically generates formulas representing the conditions that the trigger inputs need to satisfy for the program execution to go down the path. We then ask a solver (such as a decision procedure) whether the formula can be true, i.e., whether there are trigger input values which will satisfy the formula. An unsatisfiable formula indicates the path just explored is not actually feasible, and we continue to explore other paths. A satisfiable formula means we have discovered a new path which depends on trigger inputs, and the formula generated represents a trigger condition. In this case, the solver also constructs the trigger values, i.e., values for the trigger inputs necessary to execute the path of interest. We can then execute the program in a controlled environment, provide it with the discovered trigger values, and observe the trigger-based behavior. By iterating this process, we automatically explore different code paths to uncover trigger-based behaviors in the program.

In some cases the solver may not be able to return an answer to the formula within a reasonable amount of time. In this case, we simply set a timeout and go on to explore other paths. Therefore, we try to explore different branches and paths as much as possible, but do not guarantee to explore all branches or paths. As our experiments demonstrate, despite this technical difficulty in certain cases, this approach offers great practical value for automatic analysis of trigger-based behavior in real malware, and in any case, is a big step forward compared to the current manual process.

An additional technical challenge for malware analysis is that often we do not have the luxury of access to source code. Even worse, malware is often packed or obfuscated. Code packing is a technique where binary code is statically compressed to save space, and only decompressed at runtime. Obfuscation is a technique which is designed to make static analysis difficult. In either case, the code will be difficult, if not impossible, to disassemble. Thus, we need to make our approach work with only access to the binary program, and moreover, deal with binary programs which may dynamically generate code and are potentially difficult to statically analyze. To this end, we employ whole-system emulation and dynamic binary instrumentation to enable mixed concrete and symbolic execution on *binaries*. To the best of our knowledge, our system is the first to enable mixed concrete and symbolic execution on binaries (see Section 1.6).

We have implemented our approach in a system called *MineSweeper*. In our experiments, we show that our system is successful at automatically analyzing trigger-based behavior in several real world malware examples, some of which are widely spread, and some of which are packed. The total time for MineSweeper to perform the analysis is usually less than 30 minutes, which otherwise might have taken a manual process days to uncover.

Contributions. This chapter proposes that automatic analysis of trigger-based behavior is possible, and designs the first holistic approach for automatically identifying trigger-based behavior in binary programs.

- We demonstrate that automatic analysis of trigger-based behavior in malware is possible. Previous analysis was completely manual, thus any automated assistance is of great value.
- We develop techniques for mixed execution of binaries and apply them to analyzing trigger-based behavior. Previous work on mixed execution required source code [6, 14]. The ability to perform mixed execution on binaries may be of independent interest to other applications as well.
- We implement our ideas in a tool called *MineSweeper*. In particular, MineSweeper automatically: a) Detects the existence of trigger-based behavior for specified trigger types, b) Finds the trigger condition, c) Finds input values that satisfy the trigger condition, when the trigger condition can be solved, and d) Feeds the trigger values to the program, causing it to exhibit the trigger behavior, so that it may be analyzed in a controlled environment. In our experiments, the end-to-end time to perform all steps to analyze the trigger-based behavior automatically is usually less than 30 minutes.
- Minesweeper *does not need source code*, and works on unmodified binary programs. The ability to analyze binaries is absolutely necessary to be a realistic approach for malware analysis. Since we dynamically instrument code to perform mixed execution on the fly, we are also able to handle obfuscated and packed code, as demonstrated by our experiments. Also, our framework is extensible to accommodate many different trigger types.

1.2 Problem Statement and Approach Overview

In this section, we describe the overall problem of automatic trigger-based behavior analysis, and give an overview of our approach. We begin by introducing the running example we use throughout the chapter. We then introduce our terminology, and the automatic trigger-based behavior analysis problem. We then describe our approach.

Motivating Example. In Figure 1.1, we show the disassembly and source code for a typical malware worm similar to MyDoom. In this example, the `ddos` action will only be activated if the call from `GetLocalTime` returns 10:06 11/9. Thus, the `ddos` action is a trigger-based behavior which will only be triggered at this specific time.

Note that although we have provided the source code for illustrative purposes, this is not typically available to the analyst. Also, we have provided the complete disassembly, though malware is often obfuscated to prevent disassembly so such information would also not be available to the analyst. Thus, in a typical scenario, the analyst would only know the assembly instructions for runs actually executed. In addition, we have shown a relatively small example: real code is often much more complex, may contain more trigger-based branches, and often other functionality that makes it difficult to even recognize where trigger-based behavior might potentially be in the program. This raises the question: how do we reason about potential trigger-based behavior in a program automatically?

1.2.1 The Automatic Trigger-based Analysis Problem

In our problem setting, we focus on automatic discovery of trigger-based behavior when given a piece of potentially malicious code and a list of *trigger-types* of interest. Typical trigger types include the system time, system events, network and keyboard inputs, and return values from library or system calls. We call inputs from trigger-types of interest *trigger inputs.* In our running example, we assume the trigger type of interest is `GetLocalTime`, thus, the returned `systime` is the trigger input.

The program execution may take different paths depending on the values of trigger inputs. Thus, certain code paths performing malicious behaviors may only be executed if the values of trigger inputs make the program execution go down a particular path. Behaviors of such code paths are called *trigger-based behavior*. The condition that the trigger inputs need to satisfy to lead the program execution to go down a path to the trigger-based code is called the *trigger condition* for the trigger-based behavior, and the values of the trigger inputs which satisfy the trigger condition are called the *trigger values*. If we supply the trigger values as the trigger inputs, the program execution will satisfy the trigger condition and activate the trigger-based behavior which enables us to observe the trigger-based behavior in a controlled environment. Note that the trigger condition is a succinct form representing trigger values which will activate the trigger-based behavior.

In our running example, the trigger condition (from the source code) is when all 4 `if` statements are true:

```
4012b1: call   401810 <_GetLocalTime@4>
4012b6: add    $0xc,%esp
4012b9: cmpw   $0x9,0xffffffee(%ebp)
4012be: jne    40132d <_main+0xad>
4012c0: cmpw   $0xa,0xfffffff0(%ebp)
4012c5: jne    40132d <_main+0xad>
4012c7: cmpw   $0xb,0xffffffea(%ebp)
4012cc: jne    40132d <_main+0xad>
4012ce: cmpw   $0x6,0xfffffff2(%ebp)
4012d3: jne    40132d <_main+0xad>
4012d5: sub    $0xc,%esp
4012d8: push   $0x404000
4012dd: call   4017a0 <ddos>
4012e2: add    $0x10,%esp
4012e5: jmp    40132d <_main+0xad>
...
40132d: ret
```

```
SYSTEMTIME systime;
GetLocalTime(&systime);
site = ‘‘www.usenix.org’’;
if (9 == systime.wDay){
   if (10 == systime.wHour){
      if (11 == systime.wMonth){
         if (6 == systime.wMinute){
            ddos(site);
        }
     }
  }
}
```

Fig. 1.1. Our running example.

$$\text{systime.wDay} == 9 \wedge \text{systime.wHour} == 10 \wedge \text{systime.wMonth} == 11 \wedge \text{systime.wMinute} == 6$$

And the trigger value is a compound statement where the `systime` structure's `wDay` field is 9, the `wHour` field is 10, the `wMonth` field is 11, and the `wMinute` field is 6.

Problem Statement. Thus the problem of automatic trigger-based behavior analysis is when given a piece of potentially malicious code and a list of trigger types of interest, we automatically explore as many different code paths as possible to: (1) discover code paths whose execution depends on trigger inputs, (2) identify the trigger condition, (3) when possible, derive trigger values which will satisfy the trigger condition, and (4) execute the program with the trigger values to observe the trigger-based behavior in a controlled environment.

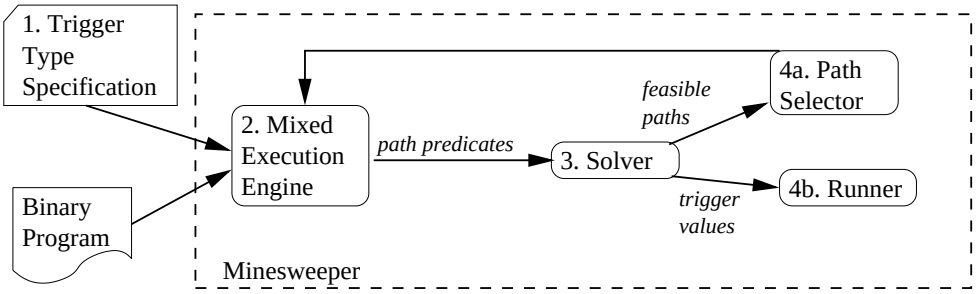

Fig. 1.2. Steps performed by MineSweeper.

1.2.2 Our Approach and System Overview

Our Approach. Since trigger-based behavior could be embedded anywhere in the program, automatically identifying trigger-based behavior requires us to automatically explore as many different execution paths that depend on trigger inputs as possible. One naïve solution would be to simply do random testing, where we could set random values to the trigger inputs and hope they will lead the program execution down different paths. However, such an approach would be hopelessly inefficient and impractical since the probability of guessing the right values to satisfy the trigger conditions would be extremely slim in most cases.

Instead, we employ an iterative approach with mixed symbolic and concrete execution, as shown in Figure 1.2. The steps are:

- Step 1: When given a malicious program, the user first selects trigger types of interest. A trigger type can be time, system events, network inputs, or any library or system call. We supply a list of trigger types that are commonly used by malware. The user can choose from the supplied list as well as define their own trigger type of interest.
- Step 2: Given the trigger types of interest, our approach then iteratively conducts mixed concrete and symbolic execution to explore the different execution paths that depend on trigger inputs and observes the trigger-based behavior. In particular, trigger inputs will be represented symbolically, and the mixed concrete and symbolic execution builds up symbolic expressions and constraints as it goes down a path. When it hits the next conditional jump depending on symbolic inputs, it will generate two *path predicates*: one for the current path continuing with the true branch, and one for the current path continuing with the false branch. The path predicate is therefore the condition on trigger inputs which make the program execution go down that path.
- Step 3: The two path predicates will then be given to a solver to see whether each formula can be satisfied, indicating whether the path is feasible. Each feasible branch will then indicate a new feasible path to be further explored. The feasible path(s) are then added to the set of paths to be further explored. For each feasible path, the solver also returns the assignment to the trigger inputs to make the formula true, i.e., the trigger values.

- Step 4.a: Our approach then selects the next path from the set of feasible paths to be further explored. The process then goes back to Step 2 to continue mixed concrete and symbolic execution along the chosen path. Execution will continue until it hits the next conditional jump that depends on trigger inputs as described in Step 2. In this manner we can force the program execution down any feasible path and thus be able to iteratively explore different execution paths depending on trigger inputs.
- Step 4.b: Our approach then executes the program concretely using the trigger values returned by the solver in Step 3, to observe the trigger-based behavior in a controlled environment.

System Overview. We have designed and implemented a system, *MineSweeper,* to realize the above approach. At a high level, MineSweeper takes as inputs the binary program to be analyzed and the trigger type specifications. MineSweeper provides a default list of trigger types commonly used in malware for the user to choose from, and also allows the user to define their own trigger types of interest. If the user does not know what trigger type the malware may use, MineSweeper can offer further assistance by monitoring for any possible inputs to the program, e.g., system calls and library calls, and then prompting the user whether the input source should be further considered as a trigger type of interest (Section 1.3.1).

MineSweeper has four components which implement the aforementioned process: the *Mixed Execution Engine*, the *Solver,* the *Path Selector,* and the *Runner,* as shown in Figure 1.2. The Mixed Execution Engine performs mixed concrete and symbolic execution and creates the path predicates. The Solver solves the path predicates to see whether they can be satisfied, and thus are feasible. For feasible paths, the Solver constructs an assignment to the input variables from the trigger types which will make the path predicates to be true. The newly discovered feasible path(s) are added to the set of paths to be further explored. The Path Selector decides which path among the set of feasible paths should be explored next. The Mixed Execution Engine then continues the mixed concrete and symbolic execution along the selected path. The constructed assignments (the trigger values) are then used as inputs to the Runner which feeds these assignments as inputs to the original program and executes the original program, thus allowing us to observe the trigger-based behavior in a controlled environment.

Note that for most malware the source code is not available. Therefore, we need to perform mixed concrete and symbolic execution on the binary directly. Previous work on mixed concrete and symbolic execution only applies to source code [6, 14]. To the best of our knowledge, no previous work could enable mixed concrete and symbolic execution directly on binaries. Even though the underlying principles between mixed execution on source code and binaries may have some parallels, mixed execution of binaries is significantly more challenging to deal with, and the actual techniques and engineering required are substantially different. At a high level, previous work on mixed execution with source code statically rewrites the program itself to perform the mixed execution. To enable mixed concrete and symbolic execution on binaries, even those that may be obfuscated or packed, we employ whole-system

emulation and dynamic binary instrumentation so that we can perform mixed concrete and symbolic execution on the fly.

1.3 MineSweeper Design

In this section, we describe the detailed design and implementation of the components in MineSweeper, including the trigger type specification, the Mixed Execution Engine, the Solver, the Path Selector, and the Runner.

1.3.1 Trigger Type Specification

The user begins analysis by specifying one or more trigger types of interest. Allowing multiple trigger types is necessary because trigger-based behavior may depend on multiple trigger types. For instance, malware may be triggered by a combination of the system time and a keyword in keyboard inputs.

By default, MineSweeper provides a list of typical trigger types commonly used in malware, including keyboard inputs, network inputs, the system clock, and other library and system calls used commonly in malware as triggers. In addition, MineSweeper is designed to be easily extensible and allows the user to add additional trigger types. For example, the user can specify any function call or system call as a trigger type.

For each trigger type that the user defines, he needs to specify where in memory the trigger inputs will be stored so that the Mixed Execution Engine can properly assign symbolic variables during mixed execution. For example, if the user specifies the return values of a new function call as a trigger type, he needs to specify where the return values are stored, e.g., in which registers, or the return memory structure of the call or call-by-reference pointers. In our running example, the specification would include that `GetLocalTime` is a trigger type. The specification would also include that `GetLocalTime` stores its results in a 16-byte structure pointed to by a stack value when `GetLocalTime` is called. During mixed execution, this information is used so that a call to `GetLocalTime` will result in a fresh symbolic variable for each byte returned. Such information is usually readily available in API documentation.

If the user does not know what trigger type the malware may use, they can configure MineSweeper to offer additional assistance. In this case, MineSweeper will monitor the program execution for possible inputs to the program, e.g., system calls and library calls. When a new input source is detected, MineSweeper prompts the user whether the input source should be considered a trigger type of interest.

1.3.2 The Mixed Execution Engine

Given the specified trigger types and the program, the Mixed Execution Engine performs mixed concrete and symbolic execution. In particular, trigger inputs are represented as symbolic variables, and the mixed execution builds up symbolic expressions and constraints on trigger inputs as it executes. When the mixed execution

encounters the next conditional jump which depends on symbolic values, it generates two path predicates representing the constraints on the trigger inputs for two new paths: one is the current path continuing with the true branch, and the other is the current path continuing with the false branch. The Mixed Execution Engine then gives both path predicates to the Solver to decide whether either one is feasible. Feasible paths are then added to the set of paths to be further explored, and the Path Selector decides which path to explore next.

In this section, we first describe how we enable mixed execution on binaries by using whole-system emulation and dynamic binary instrumentation, and then describe how we create new symbolic variables for trigger inputs. Since x86 instruction set is very complex, we convert x86 instructions to be symbolically executed to a simpler Intermediate Representation (IR) that we design, and we perform symbolic execution on the IR. Since mixed execution can be viewed at high level as achieving the same results as plain symbolic execution, but with performance enhancements, for ease of explanation, we explain first plain symbolic execution and how we generate path predicates in Section 1.3.2, and then explain how we enhance the performance by using mixed execution in Section 1.3.2.

Whole system emulation and dynamic binary instrumentation. Since for most malware we do not have access to source code, we need to perform mixed symbolic and concrete execution with only access to the program binary. Static binary instrumentation is in general considered an unsolved problem, not to mention that malware routinely use code packing and obfuscation which makes static binary instrumentation look even more hopeless. Thus, we take the approach of dynamic binary instrumentation. In particular, we build our Mixed Execution Engine on top of a whole system emulator (in our implementation, we use QEMU [2], Section 1.4.1) and perform dynamic binary instrumentation on-the-fly. By adding hooks to the emulator, our system is notified for each instruction to be executed in the original program, at which time we insert code to perform the mixed execution.

To perform mixed execution, for each instruction to be executed in the original program, we need to insert code to do two things: (1) check whether the instruction will read any trigger inputs, and if yes, we need to create new symbolic variables to represent the trigger inputs; (2) depending on the instruction, executes the instruction concretely (if all operands are concrete) or symbolically (if at least one operand is symbolic). We describe how we accomplish these two things in more detail below.

Creating New Symbolic Variables for Trigger Inputs

For each instruction to be executed in the original program, the Mixed Execution Engine first checks whether the instruction reads any inputs from the trigger types, such as I/O reads including keyboard and network inputs or returns from a function call of a trigger type. If so, the Mixed Execution Engine then assigns the locations (e.g., return registers, stack variables, etc.) from the specification fresh symbolic variables.

In the case where a function call is declared as a trigger type, when the entry point of the function call is executed, Mixed Execution Engine identifies the return

address. Then, when the function call returns the Mixed Execution Engine sets the specified buffers on the stack or the registers returning values as fresh symbolic variables. Note that this is why we require the user to provide the information about which buffer on the stack or which register contains inputs from the trigger types when the user defines a particular function as a trigger type, as mentioned in Section 1.3.1.

Symbolic Execution

At a high level, mixed concrete and symbolic execution can be viewed as achieving the same result as plain symbolic execution, but more efficiently. Thus, for ease of explanation, we explain in this section how we perform plain symbolic execution in our problem setting, and explain in Section 1.3.2 how we enhance the efficiency of plain symbolic execution using mixed concrete and symbolic execution.

Translating to an Intermediate Representation (IR). In order to perform sound symbolic execution, we must correctly interpret the semantics and effects of all assembly statements. The x86 instruction set is complex—many instructions have implicit side effects (e.g., `add` sets the `eflags` register on overflow), may have implicit operands (e.g., the memory segment selector), may behave differently for different operands (e.g., shifts by 0 do not set `eflags`), and there are even single instruction loops (e.g., `rep` instructions). Thus, to reduce the complexity of the symbolic execution logic, for each instruction that needs to be executed symbolically, we first translate it into a sequence of much simpler intermediate representation (IR) statements that we have designed. Our IR resembles a RISC-like assembly language, as shown in Table 1.1. The translation from an x86 instruction to our IR is designed to correctly model the semantics of the original x86 instruction, including making all the implicit side effects explicit (e.g., setting the `eflags` register). We then perform symbolic execution on the IR statements, instead of directly with the x86 instruction set.

Instructions i	::=	$*(r_1) := r_2 \mid r_1 := *(r_2) \mid r := v \mid r := r_1 \square_b v$
		$\mid r := \square_u v \mid$ `label` $l_i \mid$ `jmp` $\ell \mid$ `ijmp` r
		$\mid$ `if` r `jmp` ℓ_1 `else jmp` ℓ_2
Operations	$\square_b$ ::=	$+, -, *, /, \ll, \gg, \&, \mid, \oplus, ==, !=, <, \leq$ (Binary operations)
	$\square_u$ ::=	$\neg, !$ (unary operations)
Operands	v ::=	n (an integer literal) $\mid r$ (a register) $\mid \ell$ (a label)
Reg. Types	τ ::=	reg64_t | reg32_t | reg16_t | reg8_t | reg1_t (number of bits)

Table 1.1. Our RISC-like assembly IR. We convert all x86 assembly instructions into this IR.

Our IR has assignments ($r := v$), binary and unary operations ($r := r_1 \square_b v$ and $r := \square_u v$ where $\square_b$ and $\square_u$ are binary and unary operators), loading a value from

memory into a register ($r_1 := *(r_2)$), storing a value ($*r_1 := r_2$), direct jumps (jmp ℓ) to a known target label (label ℓ_i), indirect jumps to a computed value stored in a register (ijmp r), and conditional jumps (if r then jmp ℓ_1 else jmp ℓ_2). Figure 1.3 shows a small portion of the x86 assembly for our running example translated into our IR.

```
// 4012b1: call 401810 <_GetLocalTime@4>
...
// 4012b9: cmpw $0x9,0xffffffee(%ebp)
t0:=ebp+0xffffffee; t1:=*t0; t2:=0x9 ≠ t1;
// 4012be: jne 40132d <_main+0xad>
if t2 ≠ 0 jmp 40132d else jmp 4012c0;
// 4012c0: cmpw $0xa,0xfffffff0(%ebp)
t3:=ebp+0xfffffff0; t4:=*t3; t5:=0xa ≠ t4;
// 4012c5: jne 40132d <_main+0xad>
if t5 ≠ 0 jmp 40132d else jmp 4012c7;
// 4012c7: cmpw $0xb,0xffffffea(%ebp)
t6:=ebp+0xffffffea; t7:=*t6; t8:=0xb ≠ t7;
// 4012cc: jne 40132d <_main+0xad>
if t8 ≠ 0 jmp 40132d else jmp 4012ce;
// 4012ce: cmpw $0x6,0xfffffff2(%ebp)
t9:=ebp+0xfffffff2; t10:=*t9; t11:=0x6 ≠ t10;
// 4012d3: jne 40132d <_main+0xad>
if t11 ≠ 0 jmp 40132d else jmp 4012d5;
// 4012d5: execute ddos code
// 40132d: do not execute ddos code
```

Fig. 1.3. The IR for the running example.

Symbolic Execution At a high level, symbolic execution builds up symbolic expressions for variables (such as registers and memory). In our setting, symbolic execution builds a path predicate for a chosen path, i.e., the formula that the trigger inputs need to satisfy in order for the code execution to go down that path. Intuitively, each conditional jump depending on trigger inputs along the chosen path places a constraint on the trigger inputs, since the different values of the trigger inputs will make the conditional jump go one way or the other. The path predicate is simply a conjunction of all these constraints.

We generate the symbolic formulas on-the-fly in a syntax-directed manner. Symbolic execution was first introduced by King [17]. Below we give a brief description of how we perform symbolic execution and compute the path predicate for the chosen path in our setting.

- For binary, unary, and assignment operations we generate a `let` expression. A `let` expression binds a unique variable name to the expression computed, e.g., in Figure 1.4 the name t_0 is bound to the expression "ebp + 0xffffffee". Variable names are derived from the operand names, and renamed if necessary to be

```
let  M_i = λx.x
 if x == (ebp+0xfffffffe) then <wMonth>
 else if x ==  (ebp+0xffffffff0) then <wDay>
 else if ... else  M_{i-1} x
in
let t_0 = ebp + 0xfffffffe in
let t_1 =  M_i t_0 in
let t_2 = 0x9 ≠ t_1 in
let t_3 = ebp + 0xffffffff0 in
let t_4 =  M_i t_3 in
let t_5 = 0xa ≠ t_4 in
let t_6 = ebp + 0xfffffffea in
let t_7 =  M_i t_0 in
let t_8 = 0xb ≠ t_7 in
let t_9 = ebp + 0xffffffff2 in
let t_10 =  M_i t_9 in
let t_11 = 0x6 ≠ t_10 in
  (t_2 == 0) // wDay is 9
  ∧ (t_5 == 0) // and wHour is 10
  ∧ (t_8 == 0) // and wMonth is 11
  ∧ (t_11 == 0) // and wMinute is 6
```

Fig. 1.4. The path predicate generated.

unique. For example, in Figure 1.4 we see that each incarnation of the virtual register t is uniquely named. Also note that each variable definition is properly scoped by the preceding statements.

- We symbolically execute loads and stores using λ-abstractions [21]. A store creates a new memory, which is a new λ abstraction. A load is modeled as a λ application to mimic reading from the current memory state. The λ-abstraction acts like an array: given an address, it returns the last value written to that address. Let $\mathcal{M}_0$ represent an initial memory state. Then a store `*a := v` to memory address a with value v (in memory context $\mathcal{M}_0$) can be modeled as an if-then-else expression with argument x:

$$\mathcal{M}_1 \doteq \lambda x.\text{if } x == a \text{ then } v \text{ else } (\mathcal{M}_0\ x)$$

  This is a function which takes an argument — an address x — and returns the value associated with the address, e.g., v if $x == a$. A memory read of address a_r is performed by function application $(\mathcal{M}_i\ a_r) \doteq$ if $a_r == a$ then v else $(\mathcal{M}_{i-1}\ a_r)$. The application evaluates the if-then-else expression, returning the last-written value to the address a_r.
- When encountering a conditional jump, we generate two path predicates: one for the current path continuing with the true branch, and the other for the current path continuing with the false branch. For example, assuming the path predicate for the current path before the conditional jump is $\mathcal{F}$, for the conditional jump `if e then jmp L1 else jmp L2` we generate the path predicates $\mathcal{F} \wedge (e ==$

$0)$ for the path continuing with the true branch, and $\mathcal{F} \wedge (e \neq 0)$ for the path continuing with the false branch. The generated path predicates will be then given to the Solver.

Figure 1.4 shows the path predicate generated for reaching the call to `ddos` (with $\mathcal{M}_i$ representing the state of memory after the call to `GetLocalTime`).

Mixed Concrete and Symbolic Execution

To enhance the efficiency of symbolic execution, we evaluate any instruction whose operands are not symbolic concretely on the real processor. For example, if $x = 5 + 6; x = x + x;$, there is no reason to build "let x = 5+6 in let x_1 = $x + x$" when we can evaluate it natively and generate $x = 22$. Also, for conditional jumps which do not depend on symbolic values, then we know the direction taken does not depend on the trigger, and thus we can just execute it concretely. Concrete execution reduces the size and complexity of the formula, but can *only* be performed if we know for certain that *all* operands are concrete. Conducting both concrete and symbolic execution is called mixed execution. In our setting, trigger inputs are represented as symbolic variables, and therefore any operand only has a symbolic value if it is derived from trigger inputs. Thus, the vast majority of instructions can potentially be evaluated concretely, offering significant performance improvements over plain symbolic execution.

To enable mixed execution, for each instruction issued, we first need to decide whether each operand is symbolic or not. For registers, we maintain a register status table which indicates whether a register holds a symbolic value, and if so, the corresponding symbolic variable. The register status table is updated during symbolic execution as writes to registers happen.

Memory operands are more complex, and it is important to distinguish between memory addresses and memory contents, each of which can be either symbolic or concrete. In the simplest case, all the memory reads and writes are to concrete memory addresses. In this case, we simply maintain a data structure which remembers which memory cells contain symbolic values and the corresponding symbolic values. A read of a memory cell of a concrete address whose content is symbolic loads the corresponding symbolic value. A write of a symbolic value to a concrete memory address similarly adds an association between the symbolic value and the concrete memory index into our data structure .

Reading or writing memory with symbolic addresses require more care because we do not know exactly what memory cells may be read or written. In these cases, since we cannot say definitively that all operands are concrete, we must perform the operation symbolically. In addition, after a write to a symbolic address, we must perform any subsequent instruction that may load a value from that cell symbolically (in the worse case, all subsequent instructions). Note that this way the correctness is guaranteed since the memory operations will be modeled as λ-abstraction as described in Section 1.3.2.

Thus, memory operations on symbolic addresses, especially stores to a symbolic address, pose a potential efficiency problem (though not a correctness problem).

Since fewer instructions may be able to be executed concretely, this could increase the formula size, and potentially increase the difficulty for the Solver to solve for the formula. For example, in some cases, a read from a symbolic address may result in a case split when solving the formula: the Solver may need to create a separate formula to solve for each possible index read. Similarly, a write to a symbolic address will lead to a case split on subsequent reads since we need to consider the case where the index read coincides with the index written. We treat the Solver as a pluggable component, and can plug in the best solver capable of analyzing these situations.

However, in our tests, reads and writes with symbolic addresses happen rarely, thus the efficiency issue with memory operations on symbolic addresses currently does not prevent us from achieving results in practice from our experience. As future work, we do plan to build in the ability to reason about where the symbolic addresses might point to, i.e. alias analysis for binaries. Such reasoning is difficult since memory is treated as one contiguous array and we do not know where one object stops and another begins (unlike in source code). Although binary alias analysis is out of scope for this chapter, we have investigated how such alias analysis may be conducted [5]. We leave incorporating these ideas into our current infrastructure as future work.

1.3.3 The Solver

For each generated path predicate, the Solver checks whether it is *satisfiable*. One of three things can happen:

- The solver returns satisfiable, which means the path is feasible. In this case, the solver adds the feasible path to the set of paths to be further explored. In addition, the solver also generates an example set of input assignments, i.e., the trigger values, which will lead the program execution down the feasible path. The trigger values are then given to the Runner to concretely execute the program with the trigger values and observe the trigger-based behavior.
- The solver returns unsatisfiable. This means that the path is infeasible, i.e., no input will ever lead us down the exact specified path, and we mark the path as such.
- The solver takes too much time or memory. We do not consider this path further. Other choices are possible, e.g., increasing the time-out. One interesting possibility is to optimistically continue symbolically executing the path. If in subsequent execution we run into code that does not depend upon the trigger type, we can still concretely execute it. For example, in:

```
if(SHA1(x) == y)
  ddos()
```

  we may not be able to solve for x for the comparison to be true, but we could still optimistically execute the `ddos` code. Technically we would not know whether the path is really feasible, thus do not know whether the malicious behavior will really be exhibited in this case. However, sometimes the information about the existence of such malicious behavior in a piece of malware may still offer value to the analyst.

Note that the practical power of our system would thus depend on the power of the solver. MineSweeper is extensible; we can plug in any Solver appropriate, and our system thus can automatically benefit from any new progress on decision procedures, etc. Currently in our implementation, we use STP as the Solver [6, 12].

1.3.4 The Path Selector

The Path Selector takes as input the set of currently discovered feasible paths to be explored, and outputs the next path selected to be explored. The Path Selector can use different heuristics to decide which path to pick from the set of feasible paths. For example, it can use breadth-first search, depth-first search, or other strategies. Ideally, we would like to have a strategy to help us uncover trigger-based behavior as early as possible.

In our approach, our strategy is to explore as many conditional jumps which depend upon trigger inputs as possible. Thus, we take a BFS-like approach where we will always try and explore a trigger-dependent branch that has never been seen before revisiting loop bodies.

When MineSweeper encounters a loop, it will initially try to explore both branches of the loop header (the loop header is the conditional jump which one branch executes the loop body, and the other branch leaves the loop). This mimics executing the loop once. Additional loop iterations will be added to the end of the path selection queue. We have found this strategy the most effective at quickly uncovering malicious behavior in our real world examples.

1.3.5 Runner

The runner takes as input the trigger values and executes the program with the trigger values in a controlled environment. In our design, the Runner intercepts any calls to the specified trigger types, and replaces the returned answer with the given trigger values. Note that since each trigger input has a fresh symbolic variable in the mixed execution, we will be able distinguish which trigger values to supply for which function returns. For example, the Solver may specify different assignment values for the first and second time a function call of a trigger type returns; in this case, the Runner will feed the different trigger values according to whether it is the first or second time the relevant function returned. In our running example, suppose the Solver output that the time should be 11/9 at 10:06 (in reality, the Solver would return an assignment of values to the trigger inputs, e.g., a value for byte 1-14 of the specified trigger type). The Runner would intercept the `GetLocalTime` call and replace the 14-bytes returned with the supplied time of 11/9 10:06.

1.4 Implementation and Evaluation

1.4.1 Implementation

We have implemented the above components in C/C++ and OCaml. We use QEMU [2], a whole system emulator, as the basis for dynamic binary instruction in the Mixed Execution Engine. Our implementation consists of about 41,000 lines of code.

Mixed Execution Engine Implementation. The translation from an x86 binary to our IR is about 20,000 lines of C/C++ code and 9000 lines of OCaml. Much of the complexity arises from the various flags and status registers different instructions may set and test. We have also developed an extensive testing infrastructure to verify the translation is correct: we can translate an x86 program into our IR, then back to x86, and have it run correctly.

The concrete and symbolic execution component is much smaller, compromising about 12000 lines of C/C++ code. In our implementation, we perform Mixed Execution Engine by a) translating the instruction into our IR, b) consulting our register and memory maps (as discussed in Section 1.3.2) to decide which operands are symbolic, and c) executing the instruction either symbolically or concretely. Also, as soon as we hit a symbolic memory address, we switch to the symbolic execution mode as described in Section 1.3.2. For efficiency, we process a block of instructions at a time. For us, a block consists of all sequential statements up to the next conditional jump. We load an instruction cache in the Mixed Execution Engine, then have it perform the above operations on a block at a time.

One potential issue is that we may encounter very long concrete runs after trigger-dependent branches. In our implementation, we use timeouts if there are other paths to explore so that we can move on and explore the new paths instead of continuing along very long runs that do not demonstrate any trigger-based behavior.

Solver Implementation. We use STP [6, 12], a decision procedure well suited for bit-vector operations commonly found in assembly, as our Solver. STP can reason about any formula over a finite domain. Since our paths are of finite length, and each variable can take on a finite value, STP could, in theory, answer any question we posed to it. However, in real life, STP may run out of memory, or take too long to return an answer. We found that formulas involving modulus or division operations can substantially increase the answer time. However, overall we have found STP effective in our experience.

Path Selector Implementation. Since trigger-based behavior is branch-based, our Path Selector follows a branch-based strategy. Conceptually, in our implementation, we would do this by forking the execution of our Mixed Execution Engine at every symbolic jump that we encounter. However, due to the size and complexity of saving, managing, and restoring all the state, we simulate this behavior by simply running the Mixed Execution Engine multiple times.

As part of our implementation, we also build a control flow like graph of conditional jumps which depend on the trigger inputs to provide visual feedback to the user. This graph provides visual feedback to the analyst as to the progress of

MineSweeper. Vertices in the graph are conditional jumps which depend on the trigger inputs. The edges are the control flow relationship between such jumps. Figure 1.5 is an example of the graph generated for NetSky. By looking at the graph the analyst can get a good high level picture as to the progress of MineSweeper, the relationship among the path predicates for the trigger conditions, and the relationship among the possibly many trigger conditions themselves.

1.4.2 Evaluation

In order to test the effectiveness of our method, we have evaluated MineSweeper on real malware. Our real world examples include widely spread email worms (NetSky [16] and MyDoom [13]), DDoS tools (TFN [3]), and a keylogger (Perfect Keylogger [1]). All of our experiments were performed on a 2.8Ghz Pentium dual-core processor with 4GB of RAM. Our experiments demonstrate that our techniques are capable of automatically analyzing current real world malware examples. Our experiments also indicate that the total analysis time is quite small compared to an otherwise manual approach.

Program	Total Time	Total STP Time	Total Nodes	# Trigger Jumps	Percent Sym. Insn.
MyDoom	28 min	2.2 min	802042	11	0.00136%
NetSky	9 min	0.3 min	119097	6	0.00040%
Perfect Keylogger	2 min	$<$0.1min	4592	2	0.00508%
TFN	21 min	6.5 min	859759	14	0.00052%

Table 1.2. Our results on several real-world malware examples.

Results Summary. Table 1.2 shows the results of our experiments. In this table, the "Total Time" column is the total end-to-end experiment time for MineSweeper to analyze each malware, i.e., the time to explore all conditional branches which depend on the trigger inputs. Note that MineSweeper is an unoptimized prototype, and that subsequent optimizations will likely bring the total time down. We break out the total time spent in STP. In our experiments, we spent about 13% time on average solving the path predicates.

The "# Trigger Jumps" column counts how many conditional jumps were based on trigger inputs. This number is important because it demonstrates that a relatively small number of branches need to be explored in order to uncover the trigger-based behavior in these experiments.

We also show the percent of symbolic vs. number of concrete (x86) instructions executed. These numbers indicate that mixed execution reduces the formula a significant amount. This demonstrates that mixed execution is a promising approach.

Below we discuss each experiment in more detail.

NetSky

Win32.NetSky is a Win32 worm that spreads via email. The NetSky worm was one of the most widely spread worms of 2004. NetSky is known to have time triggered functionality, however different variants trigger at different times. For example, the C variant is triggered on February 26, 2004 between 6am and 9am [9]. The D variant is triggered on March 2, 2004, when the hour is between 6am and 8am [16]. The NetSky binary we analyzed was packed to prevent static analysis.

In our analysis, MineSweeper output that the library call `GetLocalTime` is a potential trigger type. We specified `GetLocalTime` as the trigger type, which returns a data structure that contains fields for the current month, day, year, hour, and minute. MineSweeper then automatically explored NetSky and analyzed its trigger-based behavior. Figure 1.5 shows a graph of program paths which depend on the trigger. In this graph, node 1 represents the day comparison, node 2 the month, node 3 the year, and nodes 4 through 6 check the hour. As we can see, in order to generate an attack, the date must be February 26, 2004, between 6-9am. According to the Symantec advisory, this is when NetSky.C attacks [9]. We can also see that when the time doesn't match, Netsky will loop back to the beginning and check again.

Overall, MineSweeper was able to discover and uncover the trigger-based behavior in about 9 minutes. We verified that all known trigger-based behavior was discovered.

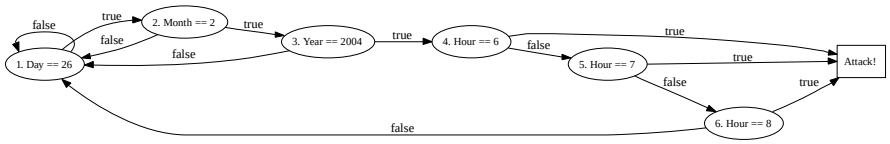

Fig. 1.5. MineSweeper generated graph showing NetSky's trigger-based behavior.

MyDoom

Win32.MyDoom [13] is another mass-mailing email worm with a built-in denial-of-service time-bomb. Different variants have different trigger dates. All variants launch DDoS attacks, most commonly against www.microsoft.com and www.sco.com. Additionally, most variants contain a termination date which causes them to stop propagating. The MyDoom binary we analyzed was packed. Overall, MineSweeper was able to discover and uncover the trigger-based behavior in MyDoom in about 28 minutes. We verified that all know trigger-based behavior was discovered.

During the initial run MineSweeper output that the library call `GetSystemTimeAsFiletime` was a potential trigger type. `GetSystemTimeAsFiletime` returns a structure which contains two 32 bit integers representing the current date and time. After

adding this specification, MineSweeper discovered MyDoom's behavior depends upon 11 different comparisons with the current date. MineSweeper automatically generated the path predicates, which STP solved. After solving these values, we were able to discover the termination date (Feb 12, 2004) as well as two DDoS dates (Feb 1 and 3, 2004). Feeding these values into the MineSweeper confirmed the DDoS. In addition, these values are confirmed by Symantec as the DDoS dates for MyDoom [13].

Perfect Keylogger

Perfect Keylogger [1] is commercial software that has the ability to trigger itself based on window title (i.e. logging is activated and deactivated by the title of the window that is the target of the keystrokes).

MineSweeper identified `GetWindowTitle` as a possible trigger type. Once we added the trigger type specification, MineSweeper discovered that Perfect Keylogger checks if the current window name contains a pre-configured key string via the `strstr` library call. In our experiment, we found that MineSweeper branched heavily in the `strstr` call, e.g., checking if the first byte of the current window name was the same as the key's first byte, then checking if the second byte of the current window name was the same as the key's second byte, etc. In this scenario, MineSweeper continued to make progress, albeit very slowly.

However, since `strstr` is a standard library function, we can be more efficient by replacing `strstr` calls with calls to a *summary function*. The summary function concisely summarizes the effects of `strstr`. Note that summary functions need only be defined once, and can be reused when analyzing other examples, and that they are a widely adopted technique in programming language research [7, 28]. Once we added this summary function, MineSweeper was able to quickly discover the trigger value in about 2 minutes. We verified that all know trigger-based behavior was discovered.

TFN: Tribe Flood Network

TFN [3] is a distributed denial-of-service attack zombie. Zombies are often found in the wild where the inner workings are unknown, e.g., the zombie may respond only to unusual messages. In the case of TFN, communication is carried out over ICMP. Different versions of TFN use different maps from command values to actions. Our goal in this experiment is to determine network inputs that would cause TFN to exhibit these different actions.

The original version of TFN that we located was Linux software. For our analysis, we have ported it to Windows since our current implementation is for Windows. Therefore, our version is not vanilla TFN, but it will still allow us to do the relevant analysis.

MineSweeper initially output that a raw ICMP network socket was the trigger type. After adding the appropriate specification, MineSweeper was able to identify and expand 14 conditional jumps that depend on network data. Using the solved

formulas that we created, we were able to determine the various command values that this version of TFN would respond to. This complex data was easily generated in only 21 minutes using the MineSweeper system.

1.5 Discussion

In this chapter, we have shown that automatically analyzing trigger-based behavior in malware is possible and described our approach and system as a first step towards this goal. In this section, we discuss lessons we learned and limitations of the current MineSweeper system.

Evasion Attacks. Identifying trigger-based behaviors in malware is an extremely challenging task. Attackers are free to make code arbitrarily hard to analyze. This follows from the fact that, at a high level, deciding whether a piece of code contains trigger-based behavior is undecidable, e.g., the trigger condition could be anything that halts the program. Thus, a tool that uncovers all trigger-based behavior all the time reduces to the halting problem.

However, this theoretic result does not mean the task of providing automatic assistance to identifying trigger-based behavior is futile. First, as our experiment results demonstrate, our system is effective in identifying trigger-based behavior in malware in the real world. Secondly, even when the attacker tries to make the code difficult to analyze, e.g., to make the formula generated difficult for the Solver to solve, our system offers value over the hopeless alternative, manual analysis. When the formulas are difficult for a Solver to solve, it is most likely that it will be even more difficult for a human to think it through in his head. In addition, the formulas generated are valuable in themselves: they concisely summarize the conditions necessary for potential trigger-based behavior which can assist in further analysis.

One popular mechanism used to thwart analysis is static binary obfuscation or run-time packing. These techniques are designed to make static analysis difficult. Since MineSweeper analyzes malware as it runs, not statically, these evasion techniques do not pose a problem to our approach, as demonstrated by our experiments.

Limitations of Current Implementation and Future Work. The current implementation of MineSweeper has a few limitations. First, system calls with symbolic arguments are difficult, as they require either a) we build a symbolic formula over the relevant code executed by the kernel, or b) create function summaries. We choose to provide summary functions to keep the size of the generated formulas manageable, thus MineSweeper only supports system calls with symbolic arguments when we have defined the appropriate function summary. Summary functions need to be specified only once, and in general are useful and are widely adopted in research.

We iteratively explore paths of finite length, thus can iteratively reason about longer and longer inputs. Handling arbitrary length inputs is a difficult problem, and usually requires (in the worse case) manually supplying program invariants. Since we have found many triggers are small and can be handled via our iterative process, we leave adding support for invariants as future work.

Finally, we currently do not handle indirect jumps dependent upon trigger values, e.g., `t = GetLocalTime; jmp t->mDay;`. In order to handle such cases, we would need to reason about the possible values for the `mDay`. This is certainly possible: we use the Solver as an oracle to enumerate possibilities, and iteratively explore them. We leave incorporating this step as future work.

As mentioned in Section 1.3.2, our original support for memory reads and writes with symbolic indexes was handled inefficiently. However, we have recently improved our system to more efficiently handle these memory accesses. This technique is described in greater detail in a later work [4].

1.6 Related Work

Time-bomb analysis. Crandall *et al.* [8] recently proposed a virtual-machine-based analysis technique to analyze the timetable of malware. Their technique uses time perturbation to identify system timers in Windows. Their technique also uses limited symbolic execution and weakest precondition calculation to identify some time-related predicates. This is a good first step towards automatic analysis of time-bombs, however, compared to our holistic approach, their technique does not follow control flow, and can only perform limited symbolic execution, not a full system mixed concrete and symbolic execution. As a result, much of their analysis done in the chapter is manual, and their techniques miss several important time-related predicates. Additionally, while their technique is specialized for time-bombs, ours is designed to support more general trigger types.

Symbolic execution. Symbolic execution was first proposed by King [17]. Recently, symbolic execution has been used for automatic test case generation [14, 22, 29], sound replay of application dialog [20], vulnerability-based signature generation [14], and program verification, e.g., ESC/Java [10, 11].

Mixed Execution. DART and EXE have proposed mixed execution for finding bugs in software and have demonstrated that this approach is effective in increasing coverage for automatic testing [6, 14]. Their work is with source code: ours is with binaries. At a high level, the approaches for mixed execution on source code and binaries are similar in spirit. However, the techniques and engineering of a solution is considerably different. For example, as mentioned one big issue is to deal with the x86 instruction set. Though this may seem like a small side issue, in reality the engineering issues are quite immense. Another difference is source code mixed execution is usually performed by rewriting the source code so that appropriate constraints are generated as it executes. For us, we must perform the instrumentation on the fly.

Moser *et al.* [19] have independently and concurrently proposed a similar method of exploring multiple paths in a binary using symbolic execution. They have also demonstrated positive results using this approach. While our approach is similar, our system is capable of handling bit-level operations and more complicated, nonlinear formulas for symbolic variables within the system.

1.7 Conclusion

We have proposed that automatically analyzing trigger-based behavior in malware is possible, and designed and implemented a system using mixed execution as a first step towards this goal. Since often trigger-based analysis of malware is manual, any help provided by MineSweeper is of great use. In our experiments with real-world malware, we demonstrate MineSweeper is capable of a) detecting the existence of trigger-based behavior for specified trigger types, b) finding the trigger condition, c) Find input values that satisfy the trigger condition, when the trigger condition can be solved, and d) feeding the trigger values to the program, causing it to exhibit the trigger-based behavior, so that it may be analyzed in a controlled environment. Even when automatic analysis fails, MineSweeper can provide an analyst with valuable information about potential trigger-based behavior: information which previously would have to be manually obtained. Automatic trigger-based behavior detection is a challenging task, and we hope our work sheds new light and encourages further work in this area.

Furthermore, this approach is specifically relevant to analysis of botnets. As discussed, botnets are merely a specific example of the general class of malicious software containing hidden behaviors. We have further demonstrated this application in other work [4].

References

1. Blazingtools perfect keylogger. `http://www.blazingtools.com/bpk.html`.
2. QEMU. `http://www.qemu.org`.
3. Tribal flood network. `http://www.cert.org/incident_notes/IN-99-07.html`.
4. D. Brumley, C. Hartwig, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically dissecting malicious binaries. `http://www.cs.cmu.edu/~chartwig/bitscope.pdf`.
5. D. Brumley and J. Newsome. Alias analysis for assembly. Technical Report CMU-CS-06-180, Carnegie Mellon University School of Computer Science, 2006.
6. C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2006.
7. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.
8. 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 *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, Oct. 2006.
9. T. L. Ferrie. Win32.Netsky.C. http://www.symantec.com/security_response/writeup.jsp?docid=2004-022417-4628-99.

10. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Estended static checking for java. In *ACM Conference on the Programming Language Design and Implementation (PLDI)*, 2002.
11. C. Flanagan and J. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th ACM Symposium on the Principles of Programming Languages (POPL)*, 2001.
12. V. Ganesh and D. Dill. STP: A decision procedure for bitvectors and arrays. `http://theory.stanford.edu/~vganesh/stp.html`.
13. S. Gettis. W32.Mydoom.B@mm. http://www.symantec.com/security_response/writeup.jsp?docid=2004-022011-2447-99.
14. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. of the 2005 Programming Language Design and Implementation Conference (PLDI)*, 2005.
15. K. Ha. Keylogger.Stawin. http://www.symantec.com/security_response/writeup.jsp?docid=2004-012915-2315-99.
16. N. Hindocha. Win32.Netsky.D. http://www.symantec.com/security_response/writeup.jsp?docid=2004-030110-0232-99.
17. J. King. Symbolic execution and program testing. *Communications of the ACM*, 19:386–394, 1976.
18. McAfee. W97M/Opey.C. `ttp://vil.nai.com/vil/content/v_10290.htm`.
19. A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*. IEEE Press, 2007.
20. J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the* 13^{th} *ACM Conference on Computer and and Communications Security (CCS)*, Oct. 2006.
21. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
22. K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for c. In *ACM SIGSOFT Sympsoium on the Foundations of Software Engineering*, 2005.
23. Symantec. Spyware.e2give. http://www.symantec.com/security_response/writeup.jsp?docid=2004-102614-1006-99.
24. Symantec. Xeram.1664. http://www.symantec.com/security_response/writeup.jsp?docid=2000-121913-2839-99.
25. United States Department of Justice Press Release. Former computer network administrator at new jersey high-tech firm sentenced to 41 months for unleashing $10 million computer "time bomb". `http://www.usdoj.gov/criminal/cybercrime/lloydSent.htm`.
26. United States Department of Justice Press Release. Former lance, inc. employee sentenced to 24 months and ordered to pay $194,609 restitution in computer fraud case. `http://www.usdoj.gov/criminal/cybercrime/SullivanSent.htm`.
27. United States Department of Justice Press Release. Former technology manager sentenced to a year in prison for computer hacking offense. `http://www.usdoj.gov/criminal/cybercrime/sheaSent.htm`.
28. Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. *ACM SIGSOFT Software Engineering Notes*, 30, 2005.
29. J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, 2006.