BitScope: Automatically Dissecting Malicious Binaries

David Brumley, Cody Hartwig, Min Gyung Kang, Zhenkai Liang
James Newsome, Pongsin Poosankam, Dawn Song, Heng Yin

March 18, 2007
Last Modified: May 23, 2007
CMU-CS-07-133

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Automatic analysis of malicious binaries is necessary in order to scale with the rapid development and recovery of malware found in the wild. The results of automatic analysis are useful for creating defense systems and understanding the current capabilities of attackers.

We propose an approach for automatic dissection of malicious binaries which can answer fundamental questions such as what behavior they exhibit, what are the relationships between their inputs and outputs, and how an attacker may be using the binary. We implement our approach in a system called BitScope. At the core of BitScope is a system which allows us to execute binaries with symbolic inputs. Executing with symbolic inputs allows us to reason about code paths without constraining the analysis to a particular input value.

We implement 5 analysis using BitScope, and demonstrate that the analysis can rapidly analyze important properties such as what behaviors the malicious binaries exhibit. For example, BitScope uncovers all commands in typical DDoS zombies and botnet programs, and uncovers significant behavior in just minutes.

This work was supported in part by CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office, the U.S. Army Research Office under the Cyber-TA Research Grant No. W911NF-06-1-0316, the ITA (International Technology Alliance), CCF-0424422, National Science Foundation Grant Nos. 0311808, 0433540, 0448452, 0627511, and by the IT R&D program of MIC(Ministry of Information and Communication)/IITA(Institute for Information Technology Advancement) [2005-S-606-02, Next Generation Prediction and Response technology for Computer and Network Security Incidents]. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the ARO, CMU, or the U.S. Government.

Keywords: symbolic execution, malware analysis, binary analysis

1 Introduction

The ability to automatically dissect a malicious binary and extract information from it is an important cornerstone for system forensic analysis and system defense. Malicious binaries, also called *malware,* include denial of service attack tools, spamming systems, worms, and botnets. New malware samples are uncovered daily through widely deployed honeypots/honeyfarms, forensic analysis of compromised systems, and through underground channels. As a result of the break-neck speed of malware development and recovery, automated analysis of malicious programs has become necessary in order to create effective defenses. Malware analysis is needed to create signatures for subsequent malware detection, detecting scams, and in general "knowing thy enemy".

Automatic dissection of malicious binaries, however, is a challenging task. There is no source code available, and to make things worse, the binary could be packed or obfuscated to evade purely static binary analysis. There may be different behavior embedded in the malicious binary which will only be activated under certain conditions such as receiving a command from the network.

Regardless of the type of malware, there are common high-level questions we would like to answer, such as:

- What actions may the malware perform, and what is the control flow between potential actions? For example, does the malware write or delete files, does it send out network packets, and does it accept remote commands?
- How do we run the malware to uncover its behavior? Since malware typically does not come with a user manual, it may be difficult to derive inputs which cause embedded behavior to be activated. For example, a malware sample may immediately exit without a particular registry key.
- How do inputs and outputs relate? For example, a DDoS client's may create a packet which in part depends upon attacker's input, and is in part constant.

Any system that can answer these questions is of high value. For example, we can use the space of possible actions to prioritize future analysis, e.g., malware that deletes files is of high importance. If we can identify unique characteristics of its behaviors, we may be able to develop signatures to weed out future malware infestations.

Unfortunately, although needed, there has to date been little progress towards useful automatic malware analysis which can answer these questions. One approach for analyzing malware is to manually use a debugger and try and reason about the behavior. Manual analysis, however, is clearly slow, error prone, and does not scale.

Another approach for malicious binary analysis is to run the malicious binary sample in a confined environment such as a virtual machine environment and observe and record its actions. Such an approach, however, can only provide very limited information. The logged information records only the external behavior of the malicious binary running in a specific setting. However, malicious binaries may have many different functionalities embedded which are only exhibited under certain environments or conditions such as when a correct command is received or a particular register key is set; and many malware will simply exit and do nothing when conditions are not met. If the virtual machine environment setup for the test does not satisfy the required conditions, the relevant malicious functionalities will not be activated. One could try to test the sample with different environment setup and try to feed random network inputs. However, setting up different environments and testing the sample with them is expensive and ineffective—the probability of guessing the right environment to satisfy the condition can be extremely low. Thus, such an approach has extremely limited utility for automatic analysis of malicious binaries, and in many cases, may not be able to produce any useful results.

Our Approach. We propose a system, called *BitScope,* to perform automatic malware dissection. BitScope takes as input a malicious binary, and outputs information about execution paths. This information is then be used by supplemental analysis designed to answer specific questions, such as what behavior the malware exhibits, what inputs activate interesting behavior, and dependency between inputs and outputs.

BitScope dissection is not performed by executing the malware with different concrete input values. Instead, BitScope abstracts away specific concrete inputs by executing the program on *symbolic* inputs which simultaneously capture a multitude of different inputs to the program. Executing with symbolic inputs allows us to reason about code paths without constraining the analysis to a particular input value.

BitScope employs whole system emulation in order to intercept any input to the program. Specific inputs are replaced with symbolic variables. BitScope then symbolically executes all instructions which are derived from an

```
struct { int type; char arg[512]; } cmd;
// Code to set up server.
while(1){
  read(net_sock, &cmd, sizeof(cmd));
  if (cmd.type == 0x1){
    DDoS(cmd.arg);
  } else if(cmd.type == 0x2){
    Spam(cmd.arg);
  } else if(cmd.type == 0x3){
    Execute(cmd.arg);
  } else {
    die();
  }
}
```

Figure 1: Our running example.

input. However, many instructions do not depend on the input, and can be executed natively. Thus, BitScope performs a mix of symbolic and concrete execution.

The BitScope approach gives us two powerful capabilities. First, the execution paths we explore are not constrained by specific inputs. As a result, we can explore a larger fraction of the program than traditional methods. Second, the information collected by BitScope can be used for additional, more specific analysis.

We build BitScope, and demonstrate its utility by creating 5 analysis components build on top of the core system. Our components can answer important questions such as what behavior the malware exhibits, what input/output dependencies exist, what inputs cause interesting behavior, and the overall flow of the program.

We test our system on two representative types of malware: botnets and DDoS zombies. Botnets and zombies serve as platforms for attackers to conduct various attacks, often with global repercussions. For example, it has been reported that a single botnet at one point used up about 15% of Yahoo's search capacity [2]; 27% of all malicious connection attempts observed in certain darknets can be directly attributed to botnet-related spreading activity [32]; and it is well known that botnets are the main sources of distributed denial-of-service (DDoS) attacks, spams, and personal information and identity thefts.

To evaluate our approach, we have run BitScope on several examples of real malware. In each case, we have observed the malware under traditional conditions and compared these observations with those given by BitScope. For each example we have been able to discover significant pieces of information that would have been very difficult or impossible to observe with traditional methods. This information includes commands accepted by various bots and zombies, capabilities of these malicious programs, and dependency information correlating their inputs to their outputs.

Contributions. We present and evaluate BitScope, a system for comprehensive analysis of malware. Specifically, we show how effective analysis can be built on top of mixed execution. Our experiments indicate that we can explore a significant portion of the malware code: in several cases we uncovered all the commands in our test DDoS zombies and bot programs. For example, our approach uncovers 709 different API call sites where malware interacts with its host — call sites such as sending out DDoS packets and executing commands on the host — while running the program alone only shows 324 call sites. Additionally, as a component of this system we present the first complete Mixed Execution Engine capable of enabling symbolic memory addresses.

2 Goals and Our Approach

2.1 Motivating Example

We motivate our work with a running example, shown in Figure 1, which shares many characteristics with typical malicious binaries. For clarity, we keep the example much simpler than most malware. We also show the malware in source-code form for ease; in practice our system works on native binaries.

Our example is a zombie which reads in a command from the network. A command consists of a command type `type` and a command argument `arg`. The example performs one of 4 different actions based on the command: it initiates a DDoS attack where the argument is the target, sends spam where the argument is the spam message, will execute a command given by the argument on the local host, or die.

2.2 Goals and Challenges to Malicious Binary Analysis

Given a malicious binary program, our goal is to automatically uncover as much about the malware as possible including what actions the binary may perform, what behaviors it may exhibit, what inputs it accepts which may activate different behaviors, and other questions.

More concretely, in our running example, we would like to learn:

- **Control flow.** For example, we would like to learn that the example implements a loop which continually reads in attackers commands from the network.
- **Behaviors.** Our example implements 4 different behaviors: a DDoS behavior, a spam behavior, a remote execution behavior, and a die behavior.
- **Inputs.** We would like to learn actual inputs to the program which activate the behaviors. For example, we are interested in what inputs will cause the program to exhibit the four different behaviors mentioned above.
- **Dependencies.** We would like to learn that the DDoS behavior is dependent upon the input supplied by the user, in specific, when the first int is 0x3, and the host is the remaining bytes. Similarly, we would like to learn that the spam messages sent are dependent upon an argument supplied by the user.

The central problem is how to automatically extract this information from the binary. One approach is to statically analyze the binary. However, static analysis of malicious binaries is impractical. First, static binary analysis in general is a hard problem. We cannot simply use source code analysis techniques as there are huge differences in scale and semantics. For example, while typical source code programs have functions, types, and local variables, assembly has instructions, does not necessarily adhere to functional abstractions even when compiled from a higher level language, has no local variables, one global memory, and a number of other problems that make source code analysis unsuitable. Our simple example compiles to over 200 lines of assembly.

Second, attackers may encrypt or pack the binary, thus hiding the actual instructions which get executed. Code packing statically compresses a binary program (or regions of a binary program). A code packer will insert an uncompress stub routine which runs at load-time. The uncompress stub uncompresses the compressed binary image in memory, then transfers control to it. Code encryption encrypts the executable segments of a binary program. A stub routine takes in a password, decrypts a stream of instructions, then executes them. The program may be completely decrypted in memory, or incrementally decrypted and executed on the fly. Code packing and encryption make static analysis difficult. While in theory we may be able to unpack the code statically then analyze it, in practice we usually do not know how the code was packed or encrypted, e.g., what algorithms were used, in the first place.

2.3 The Intuition Behind Our Approach

The intuition behind our approach is we can run the binary and collect the desired information. The main questions we must answer are how to run the program so that we can collect as much information as possible. As aforementioned, most malware has embedded behaviors which will only be activated by certain inputs. Simply running the program on random inputs is insufficient, since random inputs will likely exhibit uninteresting behavior.

Instead of using specific inputs, we run the program using symbolic inputs which stand in for a multitude of specific inputs. Any instruction which depends upon the input must then be performed symbolically. For example, the instruction `add x,y` where `x` and `y` are derived from the input creates the symbolic expression $x = x + y$, and is not restricted to specific values of x or y. Tests and conditional jumps, however, add restrictions. The true branch in the conditional jump restricts the current symbolic formula to values which are non-zero. Similarly, the false branch restricts the symbolic formula to values which are zero. Thus, symbolic execution can allow us to explore different program paths and observe the malware's behaviors under different conditions.

In order to enable the user to introduce symbolic inputs for any input source, e.g., a network input, a file descriptor, libraries, etc., we build a symbolic system environment which provides symbolic inputs to the malware as it executes.

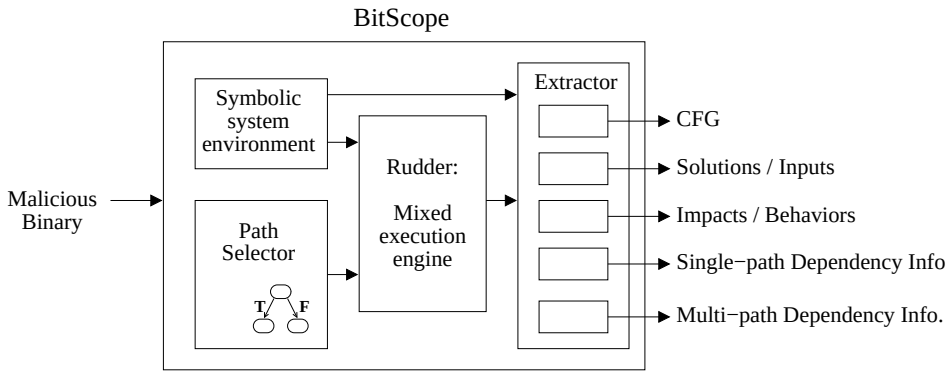


Figure 2: BitScope Architecture Overview

We perform a mixed symbolic and concrete execution on the malware, where we symbolically execute instructions which depend on the symbolic inputs, and concretely execute instructions which only depend on concrete values.

Mixed execution produces symbolic formulas in terms of the input values. The generated formulas can be used to facilitate subsequent analysis. For example, we can generate a specific input which will execute a desired path. Other analysis include driving execution down alternate code paths, reasoning about the control flow of the program, and inferring dependencies between inputs and outputs.

3 BitScope System Overview

At a high level, BitScope takes a malicious binary as input and outputs a series of analyses. This output includes: a control-flow graph of discovered code, inputs required by the binary to drive the different execution paths discovered, impact that the binary has on the system, and dependency between the inputs and outputs of the malicious binary.

Our system is composed of four components which complement each other to yield this comprehensive view of the analyzed malicious software. As shown in Figure 2, these components include: the Symbolic System Environment, Rudder, the Path Selector, and the Extractor. We give an overview for each of the components below.

The Symbolic System Environment. The Symbolic System Environment monitors the flow of information in and out of the malicious binary. Specifically, it manipulates inputs to the malicious binary to control the execution of the malicious software, and it records the outputs from the malicious binary including its impact on the system such as sending packets and writing to files. The logged information will be used by the Extractor to provide analysis results.

The Symbolic System Environment is built on top of the whole-system emulator, QEMU. At a high level, the Symbolic System Environment works by intercepting Windows API calls made by the malicious software. We do this by adding hooks to QEMU's execution.

A malicious binary receives input data whenever a Windows API call returns some information to the malicious software. When this happens, instead of allowing the actual concrete value to be returned, the Symbolic System Environment will create a new symbolic variable that represents the return value. This symbolic variable represents all the values that could have been returned to the malicious binary. Rudder, described later, uses these symbolic variables to perform symbolic execution on the malicious binary.

Information also flows out of the system when Windows API calls are made. Therefore, we log the calls that are made as well as the arguments to these calls. If the arguments to these calls are symbolic, then we determine how these symbolic outputs correspond to the original symbolic inputs.

For example, consider:

```
send(fd, buf, 30, 0);
```

In this call, the data being written to the socket is considered output from the binary. If `buf` is marked as symbolic, we will find how `buf` relates to the inputs the malicious binary has received. The above call also returns information

to the binary. Therefore, we will mark the return value as symbolic. Later, if the return value is checked for success, this will allow us to investigate what the malicious binary does both in case of success and in case of failure.

Rudder. Rudder is responsible for performing mixed concrete and symbolic execution on malicious software. Symbolic execution is necessary to handle the symbolic variables that are introduced by the Symbolic System Environment. Meanwhile, concrete execution is used as an optimization for all operations that do not depend on those symbolic variables.

As malicious software is executed, operations will be performed on the symbolic variables that the Symbolic System Environment introduces. Symbolic execution allows us to track how other values become symbolic as the program executes and to relate these derived values to the original symbolic variables that were introduced. For each newly derived symbolic value, we build a formula capturing this relation. Then, when we reach a branch in the malicious code that depends on a symbolic value, we can express the condition of that branch in terms of this formula. In particular, the constraint required to take the true or false branch, called a *path predicate* , is the logical conjunction of the constraints of all previous branches along the execution path and the constraint introduced by the current branch. Rudder builds this formula in terms of the original symbolic variables introduced by the Symbolic System Environment.

By solving the formula that Rudder has constructed, we can determine which directions of the branch are satisfiable under the current constraints. The true benefit of the Symbolic System Environment shows here. For each satisfiable path, we can find the original input values that would ordinarily drive execution down that path. Newly discovered feasible path(s) are then added to the pool of feasible paths maintained by the Path Selector to be further explored.

The Path Selector The Path Selector keeps a pool of feasible paths to be further explored.

In general, it is unreasonable to explore all the paths through a binary. In many cases, loops introduce an infinite number of such paths. Therefore, it is the job of the Path Selector to prioritize the paths that are available. Since different approaches work better in different situations, the Path Selector is modular to allow different algorithms to be dropped in. For example, if the user is able to disassemble the input binary and find parts of the code that are specifically interesting to investigate, the Path Selector can use the disassembled code to give shorter paths to the points of interest a higher priority. In other cases, where no such information is available, the path selector can prioritize paths based on a metric such as symbolic branch coverage. In this case, paths involving unvisited branches will be given priority over paths that are retracing visited code.

The Extractor The Extractor is responsible for analyzing the information that other system components obtain and providing that analysis to the user. It takes information from Rudder and the Symbolic System Environment and produces outputs such as a control-flow graph of discovered code, inputs required by the binary to drive the different execution paths discovered, impact that the binary has on the system, and dependency between the inputs and outputs of the malicious binary.

4 System Design and Implementation

In this section we describe the details of the design and implementation of the BitScope system, including the four components: Symbolic System Environment, Path Selector, Rudder, and Extractor.

4.1 Building up Symbolic System Environment

Using the Symbolic System Environment, we manipulate the environment of the malicious code. As described in Section 3, the goal of the Symbolic System Environment is two-fold; it creates symbolic inputs the malware reads in and it logs the actions of the malware.

It achieves these goals through hooking Windows API calls. These hooks are written as modular plugins to the Symbolic System Environment. Windows API calls are intercepted by adding code to QEMU that executes right before the emulated environment would jump to a hooked function. When the emulated CPU reaches the entry point

of a hooked function, the QEMU will execute the hook associated with that function. When the hook is finished, it can allow the actual API function to be executed, or simply jump over it. If the actual function is called, another hook can be dynamically added to be called when the API function returns.

Hooks can mark pieces of data in the system as symbolic. For example, when `recv` is called, we often will mark the buffer it writes into as symbolic, which allows us to trace its effects throughout the system. Hooks most commonly mark return buffers and return values as symbolic. However, hooks can be far more complex, and even maintain state of their own between calls. As described in the next section, Rudder is notified of the creation of symbolic variables by the Symbolic System Environment, which will create new symbolic variables and notify Rudder which register or memory location it represents.

Hooks that we implement also serve a second purpose. Since hooks execute in QEMU, they have access to all information about the emulated machine. For example, they can inspect memory or examine register state. This yields the ideal environment to log information about API calls that are made. When an API call is made, we note where it's called from as well as the arguments it's called with. Additionally, we make note of whether the arguments have been marked as symbolic. If they have, the information from the Symbolic System Environment and Rudder will allow the Extractor to correlate the call being made to the original symbolic system inputs. This correlation between input and output provides important dependency information and makes these logs significantly more valuable than traditional logging.

4.2 Mixed Concrete and Symbolic Execution

We use the term Mixed Execution to refer to the combination of concrete and symbolic execution. In our system, this is realized by Rudder, which is responsible for actually executing the malicious binary. Additionally, Rudder is responsible for ensuring that execution paths that we discover are realizable, i.e., that there exists real inputs to the system that will drive execution down the same paths. The concept of Mixed Execution originally proposed in EXE [10] and DART [20] for bug finding in source code. To the best of our knowledge, we are the first one to develop a fully-functional Mixed Execution Engine for binaries including enabling symbolic memory addresses. Mixed Execution on binaries although is similar in spirit to Mixed Execution on source code, however, the details of the design and implementation are completely different, and Mixed Execution on binaries is significantly more challenging as binaries in general are harder to analyze than source code.

By utilizing symbolic execution, Rudder is able to find branches in the malicious binary that depend on symbolic variables as the program executes. When this type of branch is found, Rudder will construct a path predicate for each branch direction. Each path predicate describes the constraints the symbolic inputs need to satisfy for the program execution to go down that path. The new path predicate is the conjunction of the constraints of the current path before the current branch and the constraint imposed by the current branch. Once these path predicates are constructed, Rudder will use the Solver to determine if each direction of the branch is satisfiable. All possible directions are given to the Path Selector to enqueue as future paths to be explored.

For each instruction, we perform mixed execution by following these steps. First, we must determine whether the instruction will execute concretely or symbolically, this is described in Section 4.2.1. If the instruction can be executed concretely, we simply execute on the real CPU. Otherwise, we must synchronize the symbolic machine with the concrete machine, as described in Section 4.2.2. Once the machines are synchronized, we are ready to translate the instruction to IR (Section 4.2.3) and execute it symbolically (Section 4.2.4.)

4.2.1 Determine Whether to Execute an Instruction Symbolically or Concretely.

An instruction can be executed concretely iff all operands of the instruction are concrete. Thus, deciding whether an instruction should be executed concretely or symbolically requires information about which data in the system is concrete and which is symbolic. For registers, we simply maintain a table denoting whether each register contains symbolic or concrete data. For memory, we keep a page-table type data structure that shadows each valid memory location, marking it as symbolic or concrete. This data structure allows us to efficiently track all valid memory. Every instruction executed symbolically must update this structure to reflect the propagation of symbolic data throughout the system. Additionally, when the Symbolic System Environment notifies Rudder of a new symbolic variable, the locations represented by this symbolic variable must be marked as symbolic.

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)
Types	τ	::=	reg64_t | reg32_t | reg16_t | reg8_t | reg1_t | Array of τ * τ

Table 1: Our RISC-like assembly IR. We convert all x86 assembly instructions into this IR.

If we determine that all operands of the instruction are concrete, we can simply execute the instruction concretely and continue with the next instruction. Otherwise, we must continue toward symbolic execution as described below.

4.2.2 Synchronize Machines

Mixed execution means that many instructions will be executed concretely and never be executed on the symbolic machine. Therefore, if an instruction to be symbolically executed has any concrete operands, we must update those concrete values inside the symbolic machine. In the case of registers, this is trivial– for an instruction about to be symbolically executed, we simply copy all of its concrete register operands from the real CPU to the symbolic machine. Memory accesses with concrete addresses are handled similarly. However, we also have to deal with memory accesses where the memory address itself is symbolic, which is described below.

Symbolic Memory Addresses. A symbolic memory address means that the data specifying which memory is to be read or written is itself symbolic. This means that we don't specifically know which memory location is about to be accessed.

In the case of a memory read, we know that some memory is being accessed, but because the address is symbolic, we don't know exactly which memory this is. In this case, we use the Solver to determine the range of possible values of this address. In some cases, the range that the Solver returns is too large to effectively consider. In this case, we add a constraint to the system to limit its size, therefore limiting the complexity that is introduced. In practice, we found that most symbolic memory accesses are already constrained to small ranges, making this unnecessary. For example, consider code that iterates over an array. Each access to the array is bounded by the constraints imposed by the iteration itself. Note that this is a conservative approach, meaning that all solutions found are still correct. Once a range is selected, we simply move all concrete memory values in that range into the symbolic machine.

In the case of a memory write, we apply a similar technique to find the range of addresses that *could be* written. We update the page-table type data structure mentioned before and mark that entire range as symbolic, thus all future accesses to that memory will be done with the symbolic machine.

In both of these cases, we continue correct mixed execution after the symbolic memory access.

4.2.3 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. x86 is much too complex to analyze directly. For example, parts of a register can be directly referenced and modified (e.g., `%al` references the lower 8 bits of `%eax`), there are single instuction loops (`repz` instructions), instructions with implicit side-effects (arithmetic instructions set the `eflags` register), and the semantics of each instruction may depend on the operand addressing mode (e.g., 8, 16, or 32-bit operands). Thus, we translate each x86 instruction into a simplified intermediate representation (IR). Our IR resembles a RISC-like assembly language, as shown in Table 1.

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). Memory is treated as an array of bytes. In the IR, we generate a variable `var mem:reg8_t[reg32_t]` to correspond to memory, which is an array which takes a 32-bit index, and returns an 8-bit value.

The translation from an x86 instruction to our IR is designed to model the semantics of the original x86 instruction, including all implicit side effects, register addressing modes, and other issues. We perform all symbolic execution on IR statements.

For example, the following assembly instructions add two numbers and then jump to address `0xff` if the result overflows `%ebx`, otherwise falls through is:

```
0x1. add 0x1254, %ebx
0x2. jo 0xff
0x3. ...
```

We translate this to the IR as:

```
T_32t1 = 0x1254;
// R_EBX is the variable in the IR for %ebx
T_32t0 = R_EBX + T_32t1;
// Set R_OF to 1 if there is overflow
R_OF = (1==(1& (((R_EBX⊕ (T_32t1⊕4294967295))
           &(T_32t2⊕T_32t0))≫31)));
cjmp(R_OF == 1, 0xff, 0x3)
```

where `R_OF` is the variable in the IR for the overflow flag, and the `cjmp` jumps to location 0xff if the overflow flag is set, and falls through to the next instruction otherwise.

4.2.4 Symbolic Execution

In concrete execution, a register or a memory location takes a concrete value such as integers. At a high level, symbolic execution allows registers and memory locations to contain symbolic expressions in addition to concrete values [26]. Thus, a value in a register may be an expression such as `X + Y` where `X` and `Y` are symbolic variables.

For example, if we symbolically execute the program:

```
x = y+1;
z = x *3;
k = 4+4;
mem[k] = z;
```

we produce the final values `z = (y+1)*3` and `mem[4+4] = (y+1)*3`. Note that in pure symbolic evaluation we need not evaluate the expression "4+4" to 8: integer literals can be treated just like variables.

Pure symbolic execution as described can produce formulas exponential in the size of the program, e.g. executing:

```
x1 = x0+x0; x2 = x1+x1; x3 = x2+x2;
```

produces the formula `x3 = x0+x0+x0+....+x0` where there are 8 `x0`'s. We use a variant of straight symbolic execution where common sub-expressions can be named using a `let` expression, reducing the overall size. For example, our symbolic evaluator will evaluate the above example as:

```
let x1 = x0+x0 in let x2 = x1+x1 in x2+x2
```

More specifically, our symbolic evaluator performs the following action based upon the type of each statement to execute:

- We generate `let` expressions for assignment operations. A `let` expression binds a unique variable name to the expression computed. `let` expressions avoid blowup due to substitution during symbolic evaluation as mentioned above.
- We symbolically execute loads and stores using λ-abstractions. 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.

- For conditional jumps of the form:

  `cjmp(`e`, true branch, false branch)`

  build a *path predicate* for following the true branch and a path predicate for the false branch. For example, if the expression e is `R_OF == 1`, and our current symbolic formula for `R_OF` is ϕ, then the path predicate is $\phi \wedge (\texttt{R\_OF} == 1)$ for the true branch, and $\phi \wedge (\texttt{R\_OF} \neq 1)$ for the false branch.

4.2.5 The Solver

A path predicate is a boolean function. Thus, a path predicate is either satisfiable or unsatisfiable. A satisfiable path predicate means that there is an assignment of values to symbolic variables in the path predicate which make it true. Since the symbolic variables in a path predicate are input variables, a satisfiable path predicate means there exists a set of inputs which would execute the path. Conversely, an unsatisfiable path predicate means that the path would never be executed.

We employ a Solver, such as a theorem prover or decision procedure, to check whether a path predicate is satisfiable. If a path predicate is satisfiable, the Solver returns an example solution. The example solution makes the path predicate true, which by construction is thus an input which takes us down the program path represented by the predicate. Rudder 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 [10, 19].

4.2.6 String Functions Optimizations.

In our experiments, we noticed that often the greatest complexity in our paths was introduced by string functions (`strcmp`, `strtok`, etc.). These string functions are extremely common in C code. We found them to be especially common in our examples, because most of these examples require parsing string input.

Upon investigation, we found that the complexity from these functions is not because of the formulas they create, but because of the complicated execution paths they create. In general, when we arrive at a branch that depends on symbolic data, we can pass this information to the Solver and find which directions are possible. For example, consider `strlen`, which could be implemented like this:

```
int i = 0;
while(*str++)i++;
return i;
```

In the shown code, if the input string is based on symbolic data, then clearly the returned value will depend on how that symbolic data is constrained. However, the value that is returned is never directly manipulated with symbolic data. Therefore, the return value of this function is *always* concrete. This is still completely correct. However, we are now, essentially, solving across the space of possible paths, instead of choosing paths and solving across the space of symbolic values.

In an effort to reduce this complexity, we have implemented function summaries for many of these functions. When one of these string functions would need to be executed symbolically, we instead execute our function summary which is responsible for propagating symbolic data in the same way that the original call would have. For example, in the case of `strlen`, we observe that the desired effect is for symbolic strings to have symbolic lengths. Therefore, we create a summary that will avoid actually calling `strlen` and instead return a new symbol.

It's important to note that these summary functions are only an optimization. They provide a method to reduce the number of paths through a given program. Without these summary functions our analysis is correct. However, we have observed better performance when using these functions.

After we obtain results using these summary functions, we use these values in a run with these summary functions disabled. This allows us to make several simplifying assumptions in our summary functions without losing overall correctness.

4.3 Path Selector

Simple Path Selection The Path Selector is responsible for prioritizing the paths through malicious software. In practice, the number of paths through software is very large, or even infinite. Therefore, we require algorithms that will allow us to prioritize these paths in order to find interesting paths in a reasonable amount of time. We have designed BitScope to work efficiently with many different levels of information about the malicious binary. Therefore, the Path Selector allows different algorithms to be used based on this information level.

In the worst case, the Path Selector has no *a priori* knowledge of the binary being analyzed. In this case, the Path Selector will attempt to explore as much of the unknown executable as possible. As Rudder queries the Path Selector about branches, the Path Selector will build a representation of the parts of the executable that have been explored. When future queries are received, priority is given to branches that will lead outside the currently known paths.

In a slightly improved case, we may have a binary which we are capable of at least partially disassembling. In this case, we locate points in the assembly that are interesting. These 'interesting points' include potentially malicious function calls. Given this information, the Path Selector will give paths that reach these interesting points a higher priority.

In both of these cases, the chief goal of the Path Selector is to use known information to most efficiently gather more information.

4.4 Extractor modules

We have implemented several Extractor modules in BitScope, which perform a range of useful analyses. Each module provides one more analyses. Modules can also be combined to build more complex analysis.

Control Flow Graph Module. The Control Flow Graph Module generates a control flow graph (CFG) of the analyzed binary. A CFG is useful for answering questions such as what system or library calls the binary uses, and what order the calls are used in. For example, simple control flow analysis can be used to show malware implements a server poll-accept-action loop. Control flow provides an important basis for subsequent analysis, and also gives a high level picture as to how different procedures or code segments relate.

Previous work has shown the value of determining the control flow of binaries when static disassembly is possible [28]. However, malware often decompresses, decrypts, or otherwise dynamically generates code at run time, making static disassembly impractical.

The Control Flow Graph Module creates as much of the CFG as possible via static analysis, and then continuously updates the CFG based on the dynamic execution of the binary. As Rudder finds and executes new paths, the Control Flow Graph Module adds them to the CFG. For each run, Rudder outputs which instructions were executed to the Control Flow Graph Module. We fill in the control flow graph by driving execution down different code paths, as described below. In addition, when code is dynamically decompressed, decrypted, or otherwise generated, the Control Flow Graph Module dumps the memory image and performs static whole-program control flow analysis to add the new code to the CFG.

Input Analysis Module Malware does not typically come with documentation, so learning how to properly run the malware, usually to observe its behavior, can be a daunting task. We implement an Extractor module which learns new, interesting input values by analyzing and solving the generated path predicates. For example, in our experiments we are able to learn the various input commands for bots.

One analysis provided by this module is to find the set of inputs that drive execution down a particular program path. A path predicate by design is a formula which is satisfiable for inputs that are accepted by a program path. We implement a feedback loop which solves path predicates to generate new inputs. Given the predicate ϕ, the Solver generates an example input I that satisfies ϕ and hence leads us down the corresponding program path. We then set $\phi = \phi \wedge (\neg I)$, and iterate to get a new input I', and so on.

Another analysis provided by this module is *goal-oriented* input generation, which finds an input that drives execution to a particular *target*. The input to goal-oriented input generation is a target node to reach, and the analysis generates an input which drives execution to the target node. We generate inputs for a particular goal by driving execution down a path to the goal, generating a path predicate which is then solved for the input. We implement

goal-oriented input generation using the CFG Extractor module to keep track of which paths we have explored in the malware. Note if the code is encrypted, we can still use this technique to explore previously untaken branches by specifying the branch target as the goal target.

For any node in the CFG, there are typically many paths that we can choose from. We isolate which paths could reach the target node by creating a chop of the CFG which includes only those paths from entry to the target node. We then select the shortest path from source to target, called the target path. The target path consists of a sequence of conditional jumps targets to take. The module uses Rudder to drive execution down the target path by only solving for the path predicate for the desired conditional jump targets. If the path utilizes parts of the CFG that were discovered via static analysis, the selected target path may be unrealizable due to imprecision of the static analysis; *i.e.*, no input would ever take that path. The selected path is unrealizable if we reach a point where the Solver is unable to find an input that satisfies the path predicate of the next desired conditional jump target. When that happens, we iterate with the next shortest path. When we discover a realizable path, we solve the path predicate and output the answer as the goal-oriented input.

Impact Analysis Module. We often want to know what types of behavior a particular piece of malware may exhibit. For example, we may want to know if it deletes files or sends network packets. We have implemented the Impact Analysis Module which determines behaviors dependent upon Windows API calls. We are able to detect control flow in the malicious binary that depends on API calls by using symbolic execution. By solving the constraints that Rudder constructs for this control flow, we can explore all code blocks dependent on these API calls. Using this method, we get much better code coverage than analysts simply running an executable would observe. Additionally, we log actions taken by the malware during these runs. This allows analysis of the impact this malware can have on a system.

Single-path Dependency Analysis Module. In many cases there is a straight-forward relationship between input and output behaviors. For example, a bot may accept a command to DDoS a particular host, and that host then appears in the output DDoS attack. The Single-path Dependency Analysis Module uncovers such dependencies.

We perform single-path dependence analysis on the information generated by Rudder. Instruction b is dependent upon instruction a if a computes a value that b uses. For example, one system call may set a value a subsequent system call uses, e.g., `stat` a filename, which is subsequently opened. Dependence analysis is calculated backwards: for the goal b, we calculate all instructions a which went into the calculation of b.

Multi-path Dependency Analysis Module. The Multi-path Dependency Analysis Module analyzes several runs of a program, using several different inputs, to infer additional dependencies.

We have implemented a generic *data-flow analysis* Extractor component. Dataflow analysis can be used to compute many interesting dependencies. Two useful ones we have implemented are may-constant analysis, which determines any constants used in the program, and global-value numbering, which determines if the program computes the same sub-expression several times.

May-constant analysis has proven useful for uncovering constants in packets sent out by malware. May-constant analysis determines, for each symbolic operation, if the output must be one of a set of constants, and if so, what possible constants it may be. Constant analysis is performed inductively. Any literal integer is a constant. Then, any instruction with all constant arguments is also a constant. One detail is that loops may produce a potentially infinite number of constants. Our analysis limits the number of constants to less than a pre-defined parameter k.

Global-value numbering is useful for a similar reason: we can see if observable output, or parts of the observable output, are related by an expression. The global value numbering algorithm is a bit more complicated, but essentially involves recognizing when two computed expressions are equivalent. Both global value numbering and may-constant analysis are covered in more detail in standard compilers books such as Muchnick [29].

4.5 System Implementation

We have implemented the above components in about 38,000 lines of C/C++ and OCaml code. Since we want our system to work with binaries (even when they are packed), we employ dynamic binary instrumentation in our system. In particular, we use QEMU [8], a whole system emulator that uses dynamic translation technique, as the basis for

dynamic binary instrumentation. At runtime, a block of instructions in the guest system are translated into a piece of code in the host system and then executed. This feature enables us to perform dynamic instrumentation on any code in the guest system (including code in the kernel space).

Symbolic System Environment Implementation. QEMU only provides a hardware-level view of the system, such as the state of registers and memory. For the analysis of malicious code, a software-level view is required. In particular, we want to know which process and which module (i.e., shared library or main executable) an instruction comes from. In addition, if this instruction is a call to a Windows API, we want to know which Windows API it is and its argument information.

To achieve these, we have developed a kernel module, and load it into the guest system to obtain the necessary software-level information. This kernel module is aware of the creation and deletion of processes. When a new process is created, the kernel module obtains the value of current CR3 register. As the CR3 register contains the physical address of the current process's page table, it is different (and unique) for each process. The kernel module is also aware of new modules being loaded. For each newly loaded module, the kernel module obtains its base address, and scans its exported section for the offsets of exported functions. Then we obtain the entry point of an exported function by adding the base address to its offset. All this information is passed on to the Symbolic System Environment through a predefined I/O port.

Thus, when executing a guest instruction, we can check the current CR3 to know which process is running, and we only perform instrumentation on the process(es) under analysis. At the beginning of every basic block, we check the current program counter with the entry points of Windows APIs that we are interested in. If it matches one entry, then we perform instrumentation on it. The instrumentation includes logging the function name and its argument list, and special handling for some APIs such as introducing symbolic inputs.

Obtaining the argument list of an API call requires examining the stack according to its prototype. We have developed a parsing tool to scan the API prototypes in Windows header files and automatically generate for each API a stub function that records the argument information.

Some APIs need special handling for either introducing symbolic inputs or implementing function summaries (such as for string functions). For most such APIs, the instrumentation is to simply make it return immediately with symbolic variables. For instance, when `recv` is called, we make it return immediately and mark the return value and the receiving buffer as symbolic. However, the instrumentation can be complicated for some APIs. An example is `gethostbyname`, which returns a pointer to `struct hostent` if successful, and NULL otherwise. We have to wait until it returns to mark the real content as symbolic, instead of simply marking the pointer as symbolic. In addition, in case the malware relies on the success of this call to do evil, we replace the argument of host name with "localhost" before the real invocation and recover it after return.

Currently, we have implemented hooks for all WinSock functions, some file operation functions such as Registry functions, and most functions involving time and date to introduce symbolic inputs. We have also implemented function summaries for most functions described in `string.h`.

The implementation of this component consists of about 14,000 lines of C code, excluding the stubs generated from the parsing tool.

Rudder Implementation. Rudder is implemented in about 14,000 lines of C/C++ code and 10,000 lines of OCaml. We use STP [10, 19] as our Solver in this version of Rudder. STP is a decision procedure well suited for the types of operations commonly found in the formulas Rudder constructs.

Path Selector Implementation. We have designed our system to allow easy implementation of new path selection algorithms. Currently, we have two interfaces which these algorithms can use: a linked in code interface, and a separate socket interface. In both cases the interface allows Rudder to give current path data to the path selection algorithm and receive commands in return. In our current implementation, these commands simply tell Rudder which path of a branch to select or if it should restart execution. In the future, we envision storing multiple virtual machine states, which will allows us to suspend and resume execution from any branch seen in the code.

The code interface is currently used by our main exploration algorithm. This algorithm builds a graph of the code that it has seen in current and previous runs and chooses new paths based on least traveled branches.

We have also implemented a socket interface. This allows path selection algorithms to be built and run separately from the main application. When a new path decision needs to be made, Rudder will simply send a request to the remote Path Selector which will respond with the new action. We use this interface with our second algorithm. This algorithm loads in the disassembly for a certain binary as well as a list of desired code locations. Currently, this algorithm will then give priority to shortest paths between the malicious software's entry point and the desired code location.

Extractor Implementation. Extractor modules are implemented in OCaml. We originally developed modules in C/C++, but found we could implement the same algorithms in OCaml much more concisely (and with fewer bugs thanks to strong type checking).

Our control flow graph module consists of about 1600 lines of OCaml. The CFG module produces graphs in the Graphviz DOT format. The CFG module can also compute the chop, a callgraph which depicts the callee/callers relationships, and a supergraph which shows both the instruction control flow and callee/caller relationships together.

We build the dependency analysis as dataflow analysis. The generic dataflow analysis is built on top of the CFG, and adds about 165 more lines of code. Our specific dependency analysis is about 1000 lines of OCaml code. Our implementation follows that of Muchnick [29].

We learn new inputs by querying STP [10, 19] with the generated path predicate's. About 1000 lines are devoted to translating expressions in our IR into STP.

5 Evaluation

We have evaluated BitScope with 8 representative malware samples. We get some of them from Malfease [1], and the others from collaborative researchers. We present detailed analysis results on three of them and summarize our results on all samples. In our experiments, we run BitScope on a Linux machine with a dual-core 3.2GHz Pentium CPU and 2GB RAM. Inside QEMU, we allocate 512M RAM for the guest system with Windows XP Professional installed. For each sample, we keep the system running until no more conditional jumps depending on symbolic variables are discovered for 2 minutes.

5.1 Detailed Analyses

Here we present detailed analysis results on three pieces of malware, Trin00, TFN2K, and SDBot 04b respectively.

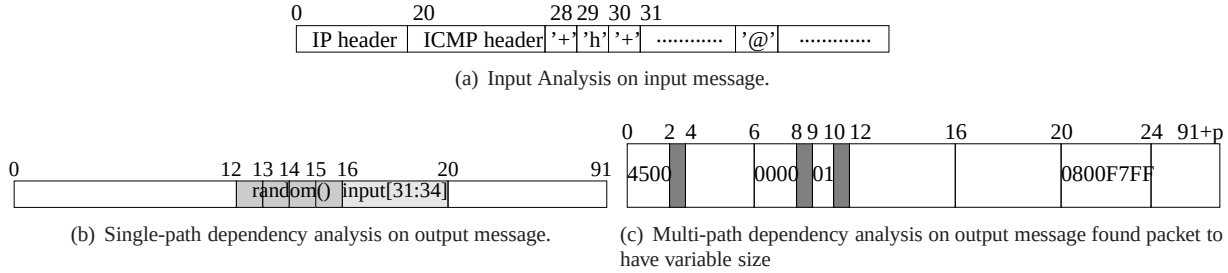
5.1.1 Trin00

Trin00 is a zombie program for launching DDoS attacks [15]. It waits for commands sent from a master and launches DDoS attacks according to the commands. Trin00 was originally a Linux zombie, but we have ported it to Windows for our analyses.

As a baseline comparison, we have executed Trin00 without BitScope, and have observed that it creates a UDP socket, binds it to port 27444, and then sends hello messages ("*hello*") to 3 IP addresses on port 31335. We believe these IP addresses to be those of the master machines. Then the program simply waits and does not exhibit further behaviors.

We then ran BitScope on the Trin00 zombie program. The end-to-end time for BitScope to analyze Trin00 took under 3 minutes. Within this time, BitScope fully explored the program and discovered 218 conditional jumps depending on symbolic inputs. BitScope is able to extract much richer information about Trin00 than the baseline case.

First, BitScope identified several network inputs which activate different behaviors in Trin00. By analyzing the information from the Symbolic System Environment and Rudder, the Input Analysis Module identified that the input message has a special format– in particular, it is composed of three parts, Input1, Input2, and Input3, separated by spaces. Further, the Input Analysis Module identified 7 specific values for Input1, "aaa", "bbb", "shi", "png", "dle", "rsz", and "xyz", which will activate different functionalities in Trin00. Essentially, Input1 provides the command. Note that the information BitScope extracted about the inputs can serve as a preliminary signature to flag potential Trin00 control traffic using the UDP port number 27444 and the 7 commands.



(a) Input Analysis on input message.

(b) Single-path dependency analysis on output message.

(c) Multi-path dependency analysis on output message found packet to have variable size

Figure 3: Analysis results on TFN2K input and output messages.

The Impact Analysis Module identified new functionalities when program execution followed the paths activated by these commands. For example, when Input1 contains the values “aaa” or “xyz”, Trin00 will continuously send out UDP packets.

The Dependency Analysis Modules output further information on the dependency between inputs and outputs. First, the Single-path Dependency Analysis Module identified that for the initial hello message sent by Trin00, the payload and destination IP addresses are independent of other symbolic inputs and the other API calls. The Multi-path Dependency Analysis Module shows that the payload and the destination IP addresses are constant across all the different runs. Second, for the UDP flooding packets generated when commands “aaa” or “xyz” are sent, the Single-path Dependency Analysis Module indicates that the destination IP address is constructed from the input message, in particular Input3, the destination port is generated from the `rand` function call[1], and the payload is independent of the inputs. To our surprise, both dependency analyses show that the payload size is a constant 4.

5.1.2 TFN2K

TFN2K is another DDoS zombie program [6]. TFN2K is more complex than Trin00, it uses raw sockets to receive commands and send flood packets. TFN2K was originally a Linux program, but for these analyses, we have ported it to Windows. Additionally, we have removed the original encryption functionality for these analyses.

First, we ran TFN2K without BitScope to observe its behaviors as the baseline case for comparison. In this case, we observed that TFN2K creates three raw sockets for ICMP, TCP and UDP respectively and waits to receive data on them. We observed no other behaviors.

We then ran TFN2K under BitScope. The end-to-end time for BitScope is under 4 minutes. Within this time, BitScope fully explored the program and discovered 20 conditional jumps depending on symbolic inputs. BitScope is able to extract much richer information about TFN2K than the baseline case.

First, the Input Analysis Module identified network inputs which activate different behaviors ini TFN2K. By analyzing the information from the Symbolic System Environment and Rudder, BitScope identified that the input message has a special format. Figure 3(a) illustrates an example of the input message format in the ICMP payload where the first and third byte (i.e. offset 28 and 30) has to be ’+’ and the second byte (i.e., offset 29) can be different values. Depending on the value of the second byte, TFN2K will perform different actions. Thus, the value of the second byte acts as a command. The Input Analysis Module uncovered twelve commands in TFN2K, which are ’a’, ’b’, ’c’, ’d’, ’e’, ’f’, ’g’, ’h’, ’i’, ’j’, ’k’, and ’l’.

The Impact Analysis Module identified new functionalities when program execution followed paths activated by these commands. For example, the command ’h’ activates an ICMP flooding attack, ’f’ activates a TCP flooding attack, and ’e’ activates a UDP flooding attack, as the Impact Analysis Module can see `sendto` being called many times in all these cases.

The only messages TFN2K sends out are flooding packets, which can be ICMP, UDP, and TCP packets, when activated by the correct command. Thus, results from the Dependency Analysis Modules are for the flooding packets. For an ICMP flooding packet, as shown in Figure 3(b), the single-path dependency analysis indicates that the source IP address is a random number from the result of the `random` function call, and the destination IP address is from

[1] BitScope hooks the `rand` function call

the input message (offset 31 to 34), which is the argument. The remaining bytes are all concrete. In the multi-path dependency analysis, we found that many fields in the packet always exhibit specific constant values. These are shown in Figure 3(c).

The Multi-path Dependency Analysis Module has identified the following special patterns in the ICMP packet generated for flooding: (1) the two bytes in offset 21 to 22 are 0x08 and 0x00, indicating that this is an ECHO request packet; (2) the two bytes in offset 23 to 24 are 0xF7 and 0xFF, which are ICMP checksum. For a UDP flooding packet, the Single-path Dependency Analysis Module show that the destination IP address is from the input, and the source IP address is randomly generated from the result of `random`. For a TCP packet, Multi-path Dependency Analysis Module shows that it always has SYN URG flags, which means this is a SYN flooding.

5.1.3 SDBot

SDBot is a typical IRC bot program that allows a remote attacker to control a computer using Internet Relay Chat (IRC) [24]. We have analyzed a specific variant: SDBot 04b.

As with our other examples, we first ran SDBot without BitScope in order to create a baseline for comparison. First we observed SDBot call `GetModuleFileName`, `GetSystemDirectory` and `CopyFile`. This resulted in SDBot copying itself to the Windows system directory. Next, SDBot calls `RegCreateKeyEx` and `RegSetValueEx` to create a registry key that causes it to start on boot. Finally, it appears to sleep between calls to `InternetGetConnectedState`.

We then ran SDBot under BitScope. SDBot is much more complex. The end-to-end time for BitScope is about 2 hours. In this time BitScope discovered 119 conditional jumps depending on symbolic inputs and new behaviors in SDBot First, the Input Analysis Module identified input messages needed to be of a particular format where an input message consists of several space-delimited strings. The first string is always the command and following strings are the arguments. The Input Analysis Module was able to extract 9 commands for IRC, and the arguments of 3 out of 9 IRC commands, “NOTICE”, “PRIVMSG”, and “332”, provide actual commands for the bot program. The Input Analysis Module identified 72 bot commands for SDBot 04b.

The Impact Analysis Module identified new behaviors when program execution followed the paths activated by the commands. For example, the “udp” bot command causes UDP flooding packets.

The Dependency Analysis Modules identified dependency information between inputs and outputs. For IRC response messages, both the Single-path Dependency Analysis Module and Multi-path Dependency Analysis Module found that part of the payload depends on the input buffer containing the command. For UDP flooding attacks, the Single-path Dependency Analysis Module shows that the payload is independent of symbolic inputs. However, the destination IP address is dependent on the command buffer and the port used can either originate from the command buffer or a simple `rand` call.

5.2 Summarized results

To demonstrate the general utility and performance of BitScope, we provide the following metrics from different aspects. The first is the end-to-end execution time. It shows how long it takes for BitScope to finish analyzing a sample. The second metric is the number of discovered conditional jumps depending on symbolic inputs, The third metric is the behaviors uncovered. Since it is in general difficult to quantify the number of different behaviors, here we use the numbers of unique call sites to Windows APIs in the sample to provide this measure. For comparison, we give the number discovered by BitScope in the third column, and the number discovered by a normal execution of the malware without BitScope in the forth column as the base-line. The difference between these two infers BitScope’s capability of discovering hidden behaviors.

We list these three metrics for all the samples in Table 2. We can see that it normally takes only minutes for BitScope to analyze a sample to uncover substantially more behavior, except SDBot 04b which took 2 hours. This execution time is still satisfactory, in the comparison with the time and efforts the human analysts spend currently. In addition, we observe that the number of unique API call sites uncovered by BitScope increases significantly than the number under the un-instrumented environment, demonstrating that BitScope is capable of revealing substantially more behaviors of malware that are invisible under normal circumstances.

	Code size	Runtime	Symbolic Cjmps Discovered	Behaviors Discovered	
				BitScope	Normal env
Trin00	201KB	569s	28	45	10
TFN2K	47KB	212s	20	39	16
SDBot 04b	238KB	≈2hr	119	115	46
evilbot	16KB	127s	7	44	22
sdbot 2311	59KB	383s	13	234	66
ircbot 0045	74KB	186s	5	86	81
ircbot 004d	34KB	181s	5	93	58
q8bot	37KB	120s	9	53	25

Table 2: Performance results

6 Discussion and Future Work

There are a variety of potential limitations to our approach and current implementation, including:

- We currently do not handle floating point numbers in the symbolic evaluator. Adding floating point numbers is straight-forward, but so far has been unnecessary. We are working to support floating point operations in future versions.
- The scalability of the solver limits the depth at which we can explore programs. Attackers could try and create formulas which would be difficult to solve. However, the formula itself still serves as a useful tool for many analysis, such as dependency analysis.
- We do not attempt to break crypto routines. Attackers often use crypto to password-protect their malware. For example, a typical scenario is a piece of malware calls `crypt` to check a password, and if the crypt'ed password matches the hard-coded password, executes the command. Although our current infrastructure does not address this problem, one technique is to recognize such calls and force execution to always succeed. We are currently implementing this approach.

Although attackers can come up with ingenous ways to make analysis hard, the advantage of our approach is we still learn about whatever we can execute.

7 Related Work

This work builds upon our previous infrastructure for mixed execution [4]. Although we use the mixed execution engine described there, we have since addressed several deficincies. In particular, we can now hook almost all windows API calls. This allows us to analyze a wider variety of software. We also now handle symbolic memory accesses and string manipulation routines. In this work, we use the mixed execution engine as a component to perform more complex analysis.

Many state-of-art dynamic analysis tools provide limited functionalities to support human analysts. Tools such as CWSandbox [35], Norman Sandbox [5], TTAnalayze [7], and Cobra [34] automatically record program actions but only on a single execution path and may miss some crucial behavior. Our technique explore multiple execution paths to address this limitation and thus have more complete view of a program's behavior. Many software testing tools also proposed to detect bugs by exploring multiple paths. For example, model checking tools [13, 23, 25] convert programs into state machine and use it to verify relevant program properties.

We use mixed symbolic and concrete execution to trigger different behaviors embedded in malware. Symbolic execution was first proposed by King in 1976 [26]. Since then it has been used in many different settings, including automatic test case generation [20, 33, 36], vulnerability-based signature generation [9], sound replay of application dialog [30], and program verification [16, 17].

EXE and DART both use mixed execution to find bugs in program source code [10, 20] while we perform mixed execution on binaries. Engineering mixed execution for binaries is quite different than for source code. For example, we must deal with symbolic memory writes and reads, which in source code is equivilant to reasoning about loading and storing pointers from collections such as arrays. Another difference includes the lack of abstractions: while source code has complex types, procedures, and variable scoping which can be used as hints for mixed execution, binaries have only simply types, no functions, only globally addressed memory region and registers.

Methods for automatic test data generation presented in [21] and [22] also use constaint solving techniques to identify interesting input values but have some limitation; the first method only works on high level languages and the latter one only handles linear constraints.

We also perform some static analysis, such as dataflow analysis. Static analysis has been used in recent researches to verify safety properties of a program [3] and to uncover malicious behaviors that may evade dynamic detection [27]. Another study uses static binary analysis to automatically generate attack signatures based on the vulnerability presented in a program [9].

We apply our technique to real-world botnet programs. Others have performed automatic malware detection [11, 12, 31], and analyzed behavior patterns in bot networks [14, 18, 32]. These approaches are complementary. For example, our approach can uncover bot commands from a bot binary, which can then be used to identify or monitor specific bot networks.

8 Conclusion

We have proposed techniques for automatically analyzing malicious binaries. We developed a system called BitScope to demonstrate our approach. At the heart of BitScope is a system for mixed execution of malicious binaries in a whole system emulation environment. The result of the mixed execution is a mathematical formula which captures the conditions necessary to execute code paths. The benefit of mixed execution is the analysis is not constrained to a specific input value, but is abstracted over all input values for a code path. We show that these formulas can be used as a basis for many interesting analysis. We demonstrate 5 such analysis, and found they produced important information for real-life malicious binaries.

References

[1] Project malfease. http://malfease.oarci.net/.

[2] http://news.bbc.co.uk/1/hi/business/6298641.stm, Jan 2007.

[3] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS '05: Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353, New York, NY, USA, 2005. ACM Press.

[4] Anonymous. Anonymized for review.

[5] N. ASA. Norman Sandbox. http://sandbox.norman.no/, 2006.

[6] J. Barlow. Tfn2k analysis. http://www.securiteam.com/securitynews/5YP0G000FS.html, Mar 2000.

[7] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[8] F. Bellard. Qemu, open source processor emulator. http://fabrice.bellard.free.fr/qemu/.

[9] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 2–16, 2006.

[10] 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.

[11] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Security and Privacy Conference*, 2005.

[12] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *Proceedings of USENIX Workshop on Steps to Reducing Unwanted Traffic on the Internet*, July 2005.

[13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 439–448, New York, NY, USA, 2000. ACM Press.

[14] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using time zones. In *Proceedings of the 13th Network and Distributed System Security Symposium (NDSS'06)*, 2006.

[15] D. Dittrich. The dos projects "trinoo" distributed denial of service attack tool. `http://staff.washington.edu/dittrich/misc/trinoo.analysis`, 1999.

[16] 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.

[17] 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.

[18] F. Freiling, T. Holz, and G. Wicherski. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. Technical Report ISSN-0935-3232, RWTH Aachen, April 2005.

[19] V. Ganesh. STP: A decision procedure for bitvectors and arrays. `http://theory.stanford.edu/~vganesh/stp.html`, 2007.

[20] 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.

[21] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *ACM Symposium on Software Testing and Analysis*, 1998.

[22] N. Gupta, A. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1998.

[23] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proceedings of the 10th SPIN Workshop on Model Checking Software (SPIN)*, 2003.

[24] T. Holz. A short visit to the bot zoo. *IEEE Security & Privacy*, 3(3):76–79, 2005.

[25] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[26] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19:386–394, 1976.

[27] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, Aug. 2005.

[28] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[29] S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[30] J. Newsome, D. Brumley, J. Franklin, and D. Song. Replayer: Automatic protocol replay by binary analysis. In R. Write, S. D. C. di Vimercati, and V. Shmatikov, editors, *In the Proceedings of the* 13^{th} *ACM Conference on Computer and and Communications Security (CCS)*, pages 311–321, 2006.

[31] M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. Semantics-based approach to malware detection. In *Proceedings of the Symposium on Principles of Programming Languages*, 2007.

[32] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *Internet Measurement Conference 2006 (IMC'06), Proceedings of*, October 2006.

[33] 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.

[34] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 264–279, Washington, DC, USA, 2006. IEEE Computer Society.

[35] C. Willems. CWSandbox: Automatic behaviour analysis of malware. `http://www.cwsandbox.org/`, 2006.

[36] 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.