Input Generation via Decomposition and Re-Stitching: Finding Bugs in Malware

Juan Caballero
CMU and UC Berkeley
jcaballero@cmu.edu

Pongsin Poosankam
CMU and UC Berkeley
ppoosank@cmu.edu

Stephen McCamant
UC Berkeley
smcc@cs.berkeley.edu

Domagoj Babić
UC Berkeley
babic@cs.berkeley.edu

Dawn Song *
UC Berkeley
dawnsong@cs.berkeley.edu

ABSTRACT

Attackers often take advantage of vulnerabilities in benign software, and the authors of benign software must search their code for bugs in hopes of finding vulnerabilities before they are exploited. But there has been little research on the converse question of whether defenders can turn the tables by finding vulnerabilities in malware. We provide a first affirmative answer to that question. We introduce a new technique, stitched dynamic symbolic execution, that makes it possible to use exploration techniques based on symbolic execution in the presence of functionalities that are common in malware and otherwise hard to analyze, such as decryption and checksums. The technique is based on decomposing the constraints induced by a program, solving only a subset, and then re-stitching the constraint solution into a complete input. We implement the approach in a system for x86 binaries, and apply it to 4 prevalent families of bots and other malware. We find 6 bugs that could be exploited by a network attacker to terminate or subvert the malware. These bugs have persisted across malware revisions for months, and even years. We discuss the possible applications and ethical considerations of this new capability.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

*This material is based upon work partially supported by the National Science Foundation under Grants No. 0311808, No. 0448452, No. 0627511, and CCF-0424422, by the Air Force Office of Scientific Research under Grant No. 22178970-4170, by the Army Research Office under grant DAAD19-02-1-0389, and by the Office of Naval Research under MURI Grant No. N000140911081. 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 Air Force Office of Scientific Research, the Army Research Office, or the Office of Naval Research.

CCS'10, October 4–8, 2010, Chicago, Illinois, USA.

Keywords

input generation, composition, malware, binary analysis

1. INTRODUCTION

Vulnerability discovery in *benign* programs has long been an important task in software security: identifying software bugs that may be remotely exploitable and creating program inputs to demonstrate their existence. However, little research has addressed vulnerabilities in *malware*. Do malicious programs have vulnerabilities? Do different binaries of the same malware family share vulnerabilities? How do we automatically discover vulnerabilities in malware? What are the implications of vulnerability discovery in malware to malware defense, law enforcement and cyberwarfare? In this paper we take the first step toward addressing these questions. In particular, we propose new symbolic reasoning techniques for automatic input generation in the presence of complex functions such as decryption and decompression, and demonstrate the effectiveness of our techniques by finding bugs in real-world malware. Our study also shows that vulnerabilities can persist for years across malware revisions. Vulnerabilities in botnet clients are valuable in many applications: besides allowing a third party to terminate or take control of a bot in the wild, they also reveal genealogical relationships between malware samples. We hope our work will spur discussions on the implications and applications of malware vulnerability discovery.

Dynamic symbolic execution techniques [24] have recently been used for a variety of input generation applications such as vulnerability discovery [7, 20, 21], automatic exploit generation [3, 23], and finding deviations between implementations [2]. By computing symbolic constraints on the input to make the program execution follow a particular path, and then solving those constraints, dynamic symbolic execution allows a system to automatically generate an input to execute a new path. Repeating this process gives an automatic exploration of the program execution space for vulnerability discovery and other applications. However, traditional dynamic symbolic execution is ineffective in the presence of certain common computation tasks, including the decryption and decompression of data, and the computation of checksums and hash functions; we call these *encoding functions*. Encoding functions result in symbolic formulas that can be difficult to solve, which is not surprising, given that cryptographic hash functions are designed to be impractical to invert [32]. Encoding functions are used widely in malware as well as benign applications. In our experiments, the traditional dynamic symbolic execution approach fails to explore the execution space of the malware samples effectively.

To address the challenges posed by the presence of encoding functions, we propose a new approach, *stitched* dynamic symbolic execution. This approach first automatically identifies potential encoding functions and their inverses (if applicable). Then, it decomposes the symbolic constraints from the execution, separating the constraints generated by each encoding function from the constraints in the rest of the execution. The solver does *not* attempt to solve the (hard) constraints induced by the encoding functions. Instead it focuses on solving the (easier) constraints from the remainder of the execution. Finally, the approach re-stitches the solver's output using the encoding functions or their inverses, creating a program input that can be fed back to the original program.

For instance, our approach can automatically identify that a particular function in an execution is performing a computation such as decrypting the input. Rather than using symbolic execution inside the decryption function, it applies symbolic execution on the outputs of the decryption function, producing constraints for the execution after the decryption. Solving those constraints generates an unencrypted message. Then, it executes the inverse (encryption) function on the unencrypted message, generating an encrypted message that can be fed back as the input to the original program.

More generally, we identify two kinds of computation that make such decomposition possible: computations that transform data into a new form that replaces the old data (such as decompression and decryption), and side computations that generate constraints that can always be satisfied by choosing values for another part of the input (such as checksums). For clarity, we explain our techniques in the context of dynamic symbolic execution, but they are equally applicable to concrete fuzz (random) testing [14,30] and taint-directed fuzzing [17].

We implement our approach in BitFuzz, a tool for automated symbolic execution of x86 binaries, implemented using our BitBlaze infrastructure [1, 42]. Our stitched dynamic symbolic execution approach applies to programs that use complex encoding functions, regardless if benign or malicious. In this paper, we use it to enable the first automated study of bugs in malware. The closest previous work we know of has focused on finding bugs on the remote administration tools that attackers use to control the malware, as opposed to the malware programs themselves, running on the compromised hosts [15, 40].

BitFuzz finds 6 new, remotely trigger-able bugs in 4 prevalent malware families that include botnet clients (Cutwail, Gheg, and MegaD) and trojans (Zbot). A remote network attacker can use these bugs to terminate or subvert the malware. We demonstrate that at least one of the bugs can be exploited, e.g., by an attacker different than the botmaster, to take over the compromised host. To confirm the value of our approach, we show that BitFuzz would be unable to find most of the bugs we report without the new techniques we introduce.

Malware vulnerabilities have a great potential for different applications such as malware removal or cyberwarfare. Some malware programs such as botnet clients are deployed at a scale that rivals popular benign applications. For instance, the recently-disabled Mariposa botnet was sending messages from more than 12 million unique IP addresses at the point it was taken down, and stole data from more than 800,000 users [26]. Our goal in this research is to demonstrate that finding vulnerabilities in widely-deployed malware such as botnet clients is technically feasible. However, the implications of the usage of malware vulnerabilities require more investigation. For example, some of the potential applications of malware vulnerabilities raise ethical and legal concerns that need to be addressed by the community. Thus, another goal of this research is to raise awareness and spur discussion in the community about the positives and negatives of the different uses of malware vulnerabilities.

In summary, this paper makes the following contributions:

- We propose a general approach, stitched dynamic symbolic execution, that incorporates techniques of identification, decomposition and re-stitching, to enable input generation in the presence of encoding functions.
- We implement our approach in BitFuzz, a tool for exploration of x86 binaries.
- Applying BitFuzz, we perform the first automated study of vulnerabilities in malware.
- We find several bugs in malware that could be triggered remotely, and verify that they persist across versions.

The remainder of this paper is organized as follows: Section 2 defines the problem we address, Section 3 describes our approach in detail, Section 4 gives additional practical details of our implementation, Section 5 describes our case studies finding bugs in malware, Section 6 discusses the implications of our results, Section 7 surveys related work, and finally, Section 8 concludes.

2. PROBLEM DEFINITION & OVERVIEW

In this section, we describe the problem we address and give an overview of our approach.

2.1 Problem Definition

Our problem is how to perform dynamic symbolic execution in the presence of encoding functions.

Background: dynamic symbolic execution. Dynamic symbolic execution [7, 20] is a technique to automatically generate inputs to explore a program's execution space. In particular, it marks the input as symbolic and performs symbolic execution along a path. The conjunction of the symbolic branch conditions forms the path predicate. By solving a modified path predicate with a solver, it automatically generates an input to make the program execution follow a new path. By repeating this process, dynamic symbolic execution can automatically find inputs to explore different execution paths of the program.

The challenge of dynamic symbolic execution with encoding functions. Often there are parts of a program that are not amenable to dynamic symbolic execution. A class of common culprits, which we call *encoding functions*, includes many instances of decryption, decompression, and checksums. For instance, consider the code in Figure 1, which is an idealized example modeled after a botnet client. A C&C message for this botnet comprises 4 bytes with the message length, followed by 20 bytes corresponding to a SHA-1 hash, followed by an encrypted payload. The bot casts the received message into a message structure, decrypts the payload using AES [10], verifies the integrity of the (decrypted) message body using the SHA-1 hash [32], and then takes a malicious action such as sending spam based on a command in the message body. Dynamic symbolic execution attempts to create a new valid input by solving a formula corresponding to the path condition for an execution path. Suppose we run the program on a message that causes the bot to participate in a DDOS attack: at a high level, the path condition takes the form

$$m' = \text{Dec}(m) \wedge h_1 = \text{SHA1}(m') \wedge m'[0] = 101 \qquad (1)$$

where m and h_1 represent two relevant parts of the program input treated as symbolic: m is the message body `m->message`, and h_1 is the message checksum `m->hash`. Dec represents the AES decryption, while SHA1 is the SHA-1 hash function. To see whether

```
1  struct msg {
2      long msg_len;
3      unsigned char hash[20];
4      unsigned char message[];
5  };
6  void process(unsigned char* network_data) {
7      int *p;
8      struct msg *m = (struct msg *) network_data;
9      aes_cbc_decrypt(m->message, m->msg_len, key);
10     p = compute_sha1(m->message, m->msg_len);
11     if (memcmp(p, m->hash, 20))
12         exit(1);
13     else {
14         int cmd = m->message[0];
15         if (cmd == 101)
16             ddos_attack(m);
17         else if (cmd == 142)
18             send_spam(m);
19           /* ... */
20     }
21 }
```

Figure 1: A simplified example of a program that uses layered input processing, including decryption (line 9) and a secure hash function for integrity verification (lines 10-12).

it can create a message to cause a different action, dynamic symbolic execution will attempt to solve the modified path condition

$$m' = \text{Dec}(m) \wedge h_1 = \text{SHA1}(m') \wedge m'[0] \neq 101 \qquad (2)$$

which differs from the original in inverting the last condition.

However, solvers tend to have a very hard time with conditions such as this one. As seen by the solver, the Dec and SHA1 functions are expanded into a complex combination of constraints that mix together the influence of many input values and are hard to reason about [12]. The solver cannot easily recognize the high-level structure of the computation, such as that the internals of the Dec and SHA1 functions are independent of the parsing condition $m'[0] \neq 101$. Such encoding functions are also just as serious an obstacle for related techniques like concrete and taint-directed fuzzing. Thus, the problem we address is how to perform input generation (such as via dynamic symbolic execution) for programs that use encoding functions.

2.2 Approach Overview

We propose an approach of *stitched* dynamic symbolic execution to perform input generation in the presence of encoding functions. We first discuss the intuition behind it, outline the steps involved, and then explain how it applies to malware vulnerability finding.

Intuition. The insight behind our approach is that it is possible to avoid the problems caused by encoding functions, by identifying and bypassing them to concentrate on the rest of the program, and re-stitching inputs using concrete execution. For instance in the path condition of formula 2, the first and second constraints come from encoding functions. Our approach can verify that they are independent from each other and the message parser (exemplified by the constraint $m'[0] \neq 101$) within the high-level structure of input processing and checking. Thus these constraints can be decomposed, and the solver can concentrate on the remainder. Solving the remaining constraints gives a partial input in the form of a value for m', and our system can then re-stitch this into a complete program input by concretely executing the encoding functions or their inverses, specifically h_1 as $\text{SHA1}(m')$ and m as $\text{Dec}^{-1}(m')$.

Stitched dynamic symbolic execution. In outline, our approach proceeds as follows. As a first phase, our approach identifies encoding functions (such as decryption and checksums) based on a program execution. Then in the second phase, our approach augments exploration based on dynamic symbolic execution by adding decomposition and re-stitching. On each iteration of exploration, we decompose the generated constraints to separate those related to encoding functions, and pass the constraints unrelated to encoding functions to a solver. The constraint solution represents a partial input; the approach then re-stitches it, with concrete execution of encoding functions and their inverses, into a complete input used for a future iteration of exploration. If as in Figure 1 there are multiple layers of encoding functions, the approach decomposes each layer in turn, and then reverses the layers in re-stitching. We detail our decomposition and re-stitching approach in Section 3.1.

Identifying encoding functions and their inverses. For identifying encoding functions, we perform a trace-based dependency analysis that is a general kind of dynamic tainting. This analysis detects functions that highly *mix* their input, i.e., an output byte depends on many input bytes. The intuition is that high mixing is what makes constraints difficult to solve. For example, a block cipher in CBC mode highly mixes its input and the constraints it introduces during decryption are hard to solve, but a stream cipher does not mix its input and thus the constraints it introduces can be easily solved. Thus, our identification technique targets encoding functions that highly mix their inputs. In addition to the encoding functions, our approach may also require their inverses (e.g., for decryption and decompression functions). The intuition behind finding inverses is that encoding functions and their inverses are often used in concert, so their implementations can often be found in the same binaries or in widely-available libraries (e.g., OpenSSL [33] or zlib [46]). In this paper, we propose a technique that given a function, identifies whether its inverse is present in a set of other functions. We detail the identification of encoding functions and their inverses in Section 3.2. We further discuss the availability of inverse functions in Section 6.2.

3. STITCHED DYNAMIC SYMBOLIC EXECUTION

In this section we describe key aspects of our approach: the conditions under which a program's constraints can be decomposed and re-stitched (Section 3.1), techniques for choosing what components' constraints to decompose (Section 3.2), and how to repeat the process when there are multiple encoding layers. (Section 3.3). An overview of the system architecture is shown in Figure 3.

3.1 Decomposition and Re-Stitching

In this section we describe the principles of our decomposition and re-stitching approach at two levels: first at the level of constraints between program values, and then more abstractly by considering a program as a collection of functional elements.

3.1.1 *Decomposing Constraints*

One perspective on decomposition is to consider a program's execution as inducing constraints among program values. These are the same constraints that are represented by formulas in symbolic execution: for instance, that one value is equal to the sum of two other values. The constraints that arise from a single program execution have the structure of a directed acyclic graph whose sources represent inputs and whose sinks represent outputs; we call this the *constraint graph*. The feasible input-output pairs for a given execution path correspond to the values that satisfy such a constraint

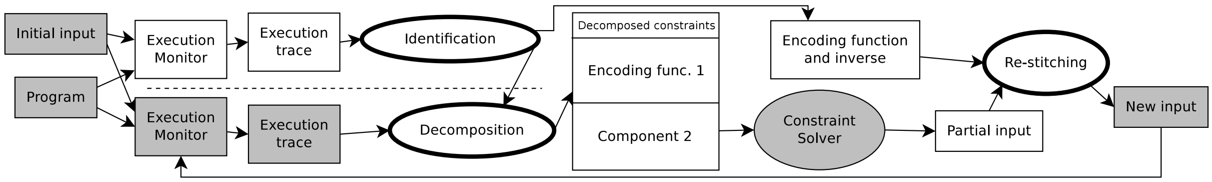

Figure 3: Architectural overview showing the parts of our decomposition-based input generation system. The steps labeled decomposition and re-stitching are discussed in Section 3.1, while identification is discussed in Section 3.2. The parts of the system shown with a gray background are the same as would be used in a non-stitching dynamic symbolic execution system. The steps above the dotted line are performed once as a setup phase, while the rest of the process is repeated for each iteration of exploration.

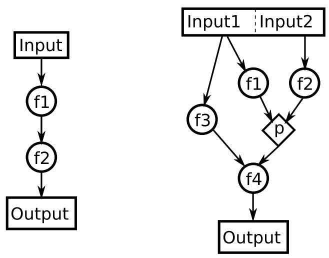

Figure 2: A graphical representation of the two styles of decomposition used in our approach. Ovals and diamonds represent computations, and edges represent the dependencies (data-flow constraints) between them. On the left is serial layering, while on the right is side-condition layering.

system, so input generation can be viewed as a kind of constraint satisfaction problem.

In this constraint-satisfaction perspective, analyzing part of a program separately corresponds to cutting the constraints that link its inputs to the rest of the execution. For a formula generated by symbolic execution, we can make part of a formula independent by renaming the variables it refers to. Following this approach, it is not necessary to extract a component as if it were a separate program. Our tool can simply perform dynamic symbolic execution on the entire program, and achieve a separation between components by using different variable names in some of the extracted constraints.

We propose two generic forms of decomposition, which are illustrated graphically in Figure 2. For each form of decomposition, we explain which parts of the program are identified for decomposition, and describe what local and global dependency conditions are necessary for the decomposition to be correct.

One set of global dependency conditions are inherent in the graph structure shown in Figure 2. If each node represents the constraints generated from one component, then for the decomposition to be correct, there must not be any constraints between values that do not correspond to edges in Figure 2. For instance the component f_2 in serial decomposition must not access the input directly.

Serial decomposition. The first style of decomposition our approach performs is between successive operations on the same information, in which the first layer is a transformation producing input to the second layer. More precisely, it involves what we call a *surjective transformation*. There are two conditions that define a surjective transformation. First, once a value has been transformed, the pre-transformed form of the input is never used again. Second, the transformation must be an onto function: every element in its codomain can be produced with some input. For example, if a function $y = x^2$ returns a signed 32-bit integer, the codomain contains 2^{32} elements. In that case, the image is a subset of the codomain that does not include for example the value -1, as it is not a possible output of the function. In Figure 2, f_1 is the component that must implement a surjective transformation. Some examples of surjective transformations include decompression and decryption. The key insight of the decomposition is that we can analyze the part of the program downstream from the transformation independently, and then simply invert the transformation to re-stitch inputs. For instance, in the example of Figure 1, the decryption operation is a surjective transformation that induces the constraint $m' = \mathrm{Dec}(m)$. To analyze the rest of the program without this encoding function, we can just rename the other uses of m' to a new variable (say m'') that is otherwise unconstrained, and analyze the program as if m'' were the input. Bypassing the decryption in this way gives

$$h_1 = \mathrm{SHA1}(m'') \wedge m''[0] = 101 \tag{3}$$

as the remaining path condition.

Side-condition decomposition. The second style of decomposition our approach performs separates two components that operate on the same data, but can still be considered mostly independent. Intuitively, a *free side-condition* is a constraint on part of a program's input that can effectively be ignored during analysis of the rest of a program, because it can always be satisfied by choosing values for another part of the input. We can be free to change this other part of the input if it does not participate in any constraints other than those from the side-condition. More precisely, a program exhibiting a free side-condition takes the form shown in the right-hand side of Figure 2. The side-condition is the constraint that the predicate p must hold between the outputs of f_1 and f_2. The side-condition is free because whatever value the first half of the input takes, p can be satisfied by making an appropriate choice for the second half of the input. An example of a free side-condition is that the checksum computed over a program's input (f_1) must equal (p) the checksum parsed from a message header (f_2).

To perform decomposition given a free side-condition, we simply replace the side-condition with a value that is always true. For instance the SHA-1 hash of Figure 1 participates in a free side-condition $h_1 = \mathrm{SHA1}(m'')$ (assuming we have already removed the decryption function as mentioned above). But h_1 does not appear anywhere else among the constraints, so we can analyze the rest of the program as if this condition were just the literal *true*. This gives the path condition:

$$\text{true} \wedge m''[0] = 101 \tag{4}$$

3.1.2 Re-Stitching

After decomposing the constraints, our system solves the constraints corresponding to the remainder of the program (excluding the encoding function(s)), as in non-stitched symbolic execution, to give a partial input. The re-stitching step builds a complete program input from this partial input by concretely execution encoding functions and their inverses. If the decomposition is correct, such a complete input is guaranteed to exist, but we construct it explicitly so that the exploration process can re-execute the program from the beginning. Once we have found a bug, a complete input confirms (independent of any assumptions about the analysis technique) that the bug is real, allows easy testing on other related samples, and is the first step in creating a working exploit.

For serial decomposition, we are given an input to f_2, and the goal is to find a corresponding input to f_1 that produces that value. This requires access to an inverse function for f_1; we discuss finding one in Section 3.2.2. (If f_1 is many-to-one, any inverse will suffice.) For instance, in the example of Figure 1, the partial input is a decrypted message, and the full input is the corresponding AES-encrypted message.

For side-condition decomposition, we are given a value for the first part of the input that is processed by f_1. The goal is to find a matching value for the rest of the input that is processed by f_2, such that the predicate p holds. For instance, in Figure 1, f_1 corresponds to the function `compute_sha1`, f_2 is the identity function copying the value `m->hash`, and p is the equality predicate. We find such a value by executing f_1 forwards, finding a value related to that value by p, and applying the inverse of f_2. A common special case is that f_2 is the identity function and the predicate p is just equality, in which case we only have to re-run f_1. For Figure 1, our tool must simply re-apply `compute_sha1` to each new message.

3.1.3 The Functional Perspective

A more abstract perspective on the decomposition our technique performs is to consider the components of the program as if they were pure functions. Of course the real programs we analyze have side-effects: a key aspect of our implementation is to automatically analyze the dependencies between operations to understand which instructions produce values that are read by other instructions. We summarize this structure to understand which operations are independent from others. In this section, we show this independence by modeling a computation as a function that takes as inputs only those values the computation depends on, and whose outputs encompass all of its side effects. This representation is convenient for formally describing the conditions that enable decomposition and re-stitching.

Serial decomposition applies when a program has the functional form $f_2(f_1(i))$ for input i, and the function f_1 (the surjective transformation) is onto: all values that might be used as inputs to f_2 could be produced as outputs of f_1 for some input. Observe that the fact that i does not appear directly as an argument to f_2 implies that f_2 has no direct dependency on the pre-transformed input. For re-stitching, we are given a partial input x_2 in $f(x_2)$, and our tool computes the corresponding full input as $x_1 = f_1^{-1}(x_2)$.

For side-condition decomposition, we say that a predicate p is a free side-condition in a program that has the functional form $f_4(f_3(i_1), p(f_1(i_1), f_2(i_2)))$, where the input is in disjoint parts i_1 and i_2. Here f_2 is a surjective transformation and p is a surjective or right-total relation: for all y there exists an x such that $p(x, y)$ is true. When p is a free side-condition, the effect of decomposition is to ignore f_1, f_2, and p, and analyze inputs i_1 as if the program were $f_4(f_3(i_1), \text{true})$. This gives a partial input x_1 for the computation $f_4(f_3(x_1), \text{true})$. To create a full input, we must also find an additional input x_2 such that $p(f_1(x_1), f_2(x_2))$ holds. Our tool computes this using the formula $x_2 = f_2^{-1}(p^{-1}(f_1(x_1)))$.

3.2 Identification

The previous section described the conditions under which decomposition is possible; we next turn to the question of how to automatically identify candidate decomposition sites. Specifically, we first discuss finding encoding functions 3.2.1, and then how to find inverses of those functions when needed 3.2.2.

3.2.1 Identifying Encoding Functions

There are two properties of an encoding function that make it profitable to use for decomposition in our approach. First, the encoding function should be difficult to reason about symbolically. Second, the way the function is used should match one of the decomposition patterns described in Section 3.1. Our identification approach is structured to check these two kinds of properties, using a common mechanism of dynamic dependency analysis.

Dynamic dependency analysis. For identifying encoding functions, we perform a trace-based dependency analysis that is a general kind of dynamic tainting. The analysis associates information with each value during execution, propagates that information when values are copied, and updates that information when values are used in an operation to give a new value. Equivalently, this can be viewed as propagating information along edges in the constraint graph (taking advantage of the fact that the execution is a topological-order traversal of that graph). Given the selection of any subset of the program state as a taint source, the analysis computes which other parts of the program state have a data dependency on that source.

Identifying high taint degree. An intuition that partially explains why many encoding functions are hard to reason about is that they mix together constraints related to many parts of the program input, which makes constraint solving difficult. For instance, this is illustrated by a contrast between an encryption function that uses a block cipher in CBC mode, and one that uses a stream cipher. Though the functions perform superficially similar tasks, the block cipher encryption is a barrier to dynamic symbolic execution because of its high mixing, while a stream cipher is not. Because of the lack of mixing, a constraint solver can efficiently determine that a single plaintext byte can be modified by making a change to the corresponding ciphertext byte. We use this intuition for detecting encoding functions for decomposition: the encoding functions we are interested in tend to mix their inputs. But we exclude simple stream ciphers from the class of encoding functions we consider, since it is easy enough to solve them directly.

We can potentially use dynamic dependency analysis to track the dependencies of values on any earlier part of the program state; for instance we have experimented with treating every input to a function as a dependency (taint) source. But for the present paper we confine ourselves to using the inputs to the entire program (i.e., from system calls) as dependency sources. To be precise our analysis assigns an identifier to each input byte, and determines, for each value in an execution, which subset of the input bytes it depends on. We call the number of such input bytes the value's *taint degree*. If the taint degree of a byte is larger than a configurable threshold, we refer to it as high-taint-degree. We group together a series of high-taint-degree values in adjacent memory locations as a single buffer; our decomposition applies to a single such buffer.

This basic technique could apply to buffers anywhere in an execution, but we further enhance it to identify functions that produce high-taint-degree buffers as output. This has several benefits: it reduces the number of candidate buffers that need to be checked in later stages, and in cases where the tool needs to later find an inverse of a computation (Section 3.2.2), it is convenient to search using a complete function. Our tool considers a buffer to be an output of a function if it is live at the point in time that a return instruction is executed. Also, to ensure we identify a function that includes the complete encoding functionality, our tool uses the dependency analysis to find the first high-taint-degree computation that the output buffer depends on, and chooses the function that encloses both this first computation and the output buffer.

In the example of Figure 1, the buffers containing the outputs of `aes_cbc_decrypt` and `compute_sha1` would both be found as candidates by this technique, since they both would contain bytes that depend on all of the input bytes (the final decrypted byte, and all of the hash value bytes).

Checking dependence conditions. Values with a high taint degree as identified above are candidates for decomposition because they are potentially problematic for symbolic reasoning. But to apply our technique to them, they must also appear in a proper context in the program to apply our decomposition. Intuitively the structure of the program must be like those in Figure 2. To be more precise, we describe (in-)dependence conditions that limit what parts of the program may use values produced by other parts of the program. The next step in our identification approach is to verify that the proper dependence conditions hold (on the observed execution). This checking is needed to avoid improper decompositions, and it also further filters the potential encoding functions identified based on taint degree.

Intuitively, the dependence conditions require that the encoding function be independent of the rest of the program, except for the specific relationships we expect. For serial decomposition, our tool checks that the input bytes that were used as inputs to the surjective transformation are not used later in the program. For side-condition decomposition, our tool checks that the result of the free side-condition predicate is the only use of the value computed from the main input (e.g., the computed checksum), and that the remaining input (e.g., the expected checksum from a header) is not used other than in the free side-condition. Our tool performs this checking using the same kind of dynamic dependency analysis used to measure taint degree.

In the example of Figure 1, our tool checks that the encrypted input to `aes_cbc_decrypt` is not used later in the program (it cannot be, because it is overwritten). It also checks that the hash buffer pointed to by h is not used other than in the `memcmp` on line 11, and that the buffer `m->hash`, containing the expected hash value, is not used elsewhere.

Identifying new encoding functions. The identification step may need to be run in each iteration of the exploration because new encoding functions functions may appear that had not been seen in previous iterations. As an optimization, BitFuzz runs the identification on the first iteration of the exploration, as shown in Figure 3, and then, on each new iteration, it checks whether the solver times out when solving any constraint. If it does, it re-runs the identification on the current execution trace.

A graph-based alternative. Our taint-degree dependency analysis can be seen as simple special case of a broader class of algorithms that identify interesting parts of a program from the structure of its data dependency (data-flow) graph. The approach we currently use has efficiency and simplicity advantages because it can operate in one pass over a trace, but in the future we are also interested in exploring more general approaches that explicitly construct the dependency graph. For instance, the interface between the two stages in a serial decomposition must be a cut in the constraint graph, and we would generally expect it to be minimal cut in the sense of the subset partial order. So we can search for candidate serial decompositions by using a maximum-flow-minimum-cut algorithm as in McCamant and Ernst's Flowcheck tool [28].

3.2.2 *Identifying Inverse Functions*

Recall that to re-stitch inputs after serial decomposition, our approach requires the inverses of surjective transformation functions. This requirement is reasonable because surjective functions like decryption and decompression are commonly the inverses of other functions (encryption and compression) that apply to arbitrary data. These functions and their inverses are often used in concert, so their implementations can often be found in the same binaries or in publicly available libraries (e.g., [33, 46]). Thus, we locate relevant inverse functions by searching for them in the code being analyzed as well as in publicly available libraries.

Specifically, we check whether two functions are each others' inverses by random testing. If f and f' are two functions, and for several randomly-chosen x and y, $f'(f(x)) = x$ and $f(f'(y)) = y$, then f and f' are likely inverses of each other over most of their domains. Suppose f is the encoding function we wish to invert. Starting with all the functions from the same binary module that were exercised in the trace, we infer their interfaces using our previous BCR tool [4]. To prioritize the candidates, we use the intuition that the encryption and decryption functions likely have similar interfaces. For each candidate inverse g, we compute a 4-element feature vector counting how many of the parameters are used only for input, only for output, or both, and how many are pointers. We then sort the candidates in increasing order of the Manhattan distances (sum of absolute differences) between their features and those of f.

For each candidate inverse g, we execute $f \circ g$ and $g \circ f$ on k random inputs each, and check whether they both return the original inputs in all cases. If so, we consider g to be the inverse of f. To match the output interface of g with the input interface of f, and vice-versa, we generate missing inputs either according to the semantics inferred by BCR (such as buffer lengths), or randomly; if there are more outputs than inputs we test each possible mapping. Increasing the parameter k improves the confidence in resulting identification, but the choice of the parameter is not very sensitive: test buffers have enough entropy that even a single false positive is unlikely, but since the tests are just concrete executions, they are inexpensive. If we do not find an inverse among the executed functions in the same module, we expand the search to other functions in the binary, in other libraries shipped with the binary, and in standard libraries.

For instance, in the example of Figure 1, our tool requires an AES encryption function to invert the AES decryption used by the bot program. In bots it is common for the encryption function to appear in the same binary, since the bot often encrypts its reply messages with the same cipher, but in the case of a standard function like AES we could also find the inverse in a standard library like OpenSSL [33].

Once an inverse function is identified, we use our previous BCR tool to extract the function [4]. The hybrid disassembly technique used by BCR extracts the body of the function, including instructions that did not appear in the execution, which is important because when re-stitching a partial input branches leading to those, previously unseen, instructions may be taken.

3.3 Multiple Encoding Layers

If a program has more than one encoding function, we can repeat our approach to decompose the constraints from each encoding function in turn, creating a multi-layered decomposition. The decomposition operates from the outside in, in the order the encoding functions are applied to the input, intuitively like peeling the layers of an onion. For instance, in the example of Figure 1, our tool decomposes first the decryption function and then the hash-checking function, finally leaving only the botnet client's command parsing and malicious behavior for exploration.

4. IMPLEMENTATION

In this section we provide implementation details for our BitFuzz tool and describe our Internet-in-a-Workstation environment.

4.1 BitFuzz

We have implemented our approach in a tool called BitFuzz. BitFuzz's operation is similar to previous exploration tools for program binaries such as SAGE [21], SmartFuzz [31], and Elcano [5], but with the addition of our stitched dynamic symbolic execution techniques. BitFuzz shares some underlying infrastructure with our previous tools including Elcano, but it lacks support for protocol information and adds other new features such as distributed operation on computer clusters.

BitFuzz is implemented using the BitBlaze [42] platform for binary analysis, which includes TEMU, an extensible whole-system emulator that implements taint propagation, and Vine, an intermediate language and analysis library that represents the precise semantics of x86 instructions in terms of a few basic operations. BitFuzz uses TEMU to collect execution traces and Vine to generate a symbolic representation of the program's computations and path condition. To solve modified path conditions, the experiments in this paper use STP [16], a complete decision procedure incorporating the theories of arrays and bit-vectors.

BitFuzz maintains two pools, of program inputs and execution traces: each input gives a trace, and each trace can yield one or more new inputs. To bias this potentially unbounded feedback towards interesting paths, it performs a breadth-first search (i.e., changing a minimal number of branches compared to the original input), prioritizes traces that cover the most new code blocks, and only reverts one occurrence of a loop condition.

Vulnerability detection. BitFuzz supports several techniques for vulnerability detection and reports any inputs flagged by these techniques. It detects program termination and invalid memory access exceptions. Executions that exceed a timeout are flagged as potential infinite loops. It also uses TEMU's taint propagation module to identify whether the input (e.g., network data) is used in the program counter or in the size parameter of a memory allocation.

Decomposition and re-stitching details. Following the approach introduced in Section 3.1.1, our system implements decomposition by making local modifications constraints generated from execution, with some additional optimizations. For serial decomposition, it uses a TEMU extension mechanism called a hook to implement the renaming of symbolic values. As a further optimization, the hook temporarily disables taint propagation inside the encoding function so that no symbolic constraints are generated. To save the work of recomputing a checksum on each iteration in the case of side-condition decomposition, our tool can also directly force the conditional branch implementing the predicate p to take the same direction it did on the original execution.

4.2 Internet-in-a-Workstation

We have developed an environment where we can run malware in isolation, without worrying about malicious behavior leaking to the Internet. Many malware programs, e.g., bots, act as network clients that start connections to remote C&C servers. Thus, the input that BitFuzz needs to feed to the program in each iteration is often the response to some request sent by the program.

All network traffic generated by the program, running in the execution monitor, is redirected to the local workstation in a manner that is transparent to the program under analysis. In addition, we have developed two helper tools: a modified DNS server which can respond to any DNS query with a preconfigured or randomly generated, IP address, and a generic replay server. The generic replay server takes as input an XML file that describes a network dialog as an ordered sequence of connections, where each connection can comprise multiple messages in either direction. It also takes as input the payload of the messages in the dialog. Such generic server simplifies the task of setting up different programs and protocols. Given a network trace of the communication we generate the XML file describing the dialog to explore, and give the replay server the seed messages for the exploration. Then, at the beginning of each exploration iteration BitFuzz hands new payload files (i.e., the re-stitched program input) to the replay server so that they are fed to the network client program under analysis when it opens a new connection.

5. EXPERIMENTAL EVALUATION

This section evaluates our approach by finding bugs in malware that uses complex encoding functions. It demonstrates that our decomposition and re-stitching approach finds some bugs in malware that would not be found without it, and that it significantly increases the efficiency of the exploration in other cases. It presents the malware bugs we find and shows that these bugs have persisted in the malware families for long periods of time, sometimes years.

Malware samples. The first column of Table 1 presents the four popular families of malware that we have used in our evaluation. Three of them (Cutwail, Gheg, and MegaD) are spam bots, while Zbot is a trojan used for stealing private information from compromised hosts. At the time of writing MegaD accounts for over 15% of the spam in the Internet, Cutwail/Pushdo for over 7% [27]. Gheg is a smaller spam contributor but is still significant with an estimated size over 60,000 bots [22].

All four malware families act as network clients, that is, when run they attempt to connect to a remote C&C server rather than opening a listening socket and await for commands. All four of them use encryption to obfuscate their network communication, avoid signature-based NIDS detection, and make it harder for analysts to reverse-engineer their C&C protocol. Cutwail, Gheg, and MegaD use proprietary encryption algorithms, while Zbot uses the well-known RC4 stream cipher. In addition to encryption, Zbot also uses an MD5 cryptographic hash function to verify the integrity of a configuration file received from the server.

Experimental setup. For each bot we are given a network trace of the bot communication from which we extract an XML representation of the dialog between the bot and the C&C server, as well as the payload of the network packets in that dialog. This information is needed by the replay server to provide the correct sequence of network packets to the bot during exploration. For example, this is needed for MegaD where the response sent by the replay server comprises two packets that need to be sent sequentially but cannot be concatenated together due to the way that the bot reads from the socket. As a seed for the exploration we use the same content observed in the dialog captured in the network trace. Other seeds can alternatively be used. Although our setup can support exploring multiple connections, currently, we focus the exploration on the first connection started by the bot.

For the experiments we run BitFuzz on a 3GHz Intel Core 2 Duo Linux workstation with 4GB of RAM running Ubuntu Server 9.04.

Name	Program size (KB)	Input size (bytes)	# Instruction ($\times 10^3$)	Decryption		Checksum/hash		Runtime (sec)
				Algorithm	Max. taint degree	Algorithm	Max. taint degree	
Zbot	126.5	5269	1307.3	RC4-256	1	MD5	4976	92
MegaD	71.0	68	4687.6	64-bit block cipher	8	none	n/a	105
Gheg	32.0	271	84.5	8-bit stream cipher	128	none	n/a	5
Cutwail	50.0	269	23.1	byte-based cipher	1	none	n/a	2

Table 1: Summary of the applications on which we performed identification of encoding functions.

The emulated guest system where the malware program runs is a Microsoft Windows XP SP3 image with 512MB of emulated RAM.

5.1 Identification of Encoding Functions and Their Inverses

The first step in our approach is to identify the encoding functions. The identification of the encoding functions happens on the execution trace produced by the seed at the beginning of the exploration. We set the taint degree threshold to 4, so that any byte that has been generated from 5 or more input bytes is flagged. Table 1 summarizes the results. The identification finds an encoding function in three of the four samples: Gheg, MegaD, and Zbot. For Cutwail, no encoding function is identified. The reason for this is that Cutwail's cipher is simple and does not contain any mixing of the input, which is the property that our encoding function identification technique detects. Without input mixing the constraints generated by the cipher are not complex to solve. We show this in the next section. In addition, Cutwail's trace does not contain any checksum functions.

For Zbot, the encoding function flagged in the identification corresponds to the MD5 checksum that it uses to verify the integrity of the configuration file it downloads from the C&C server. In addition to the checksum, Zbot uses the RC4 cipher to protect its communication, which is not flagged by our technique. This happens because RC4 is a stream cipher that does no mixing of the input, i.e., it does not use input or output bytes to update its internal state. The input is simply combined with a pseudo-random keystream using bit-wise exclusive-or. Since the keystream is not derived from the input but from a key in the data section, it is concrete for the solver. Thus, the solver only needs to invert the exclusive-or computation to generate an input, which means that RC4 introduces no hard-to-solve constraints.

For the other two samples (Gheg and MegaD) the encoding function flagged by the identification corresponds to the cipher. MegaD uses a 64-bit block cipher, which mixes 8 bytes from the input before combining them with the key. Gheg's cipher uses a one-byte key that is combined with the first input byte to produce a one-byte output that is used also as key to encode the next byte. This process repeats and the mixing (taint degree) of each new output byte increases by one. Neither Gheg nor MegaD uses a checksum.

Once the encoding functions have been identified, BitFuzz introduces new symbols for the outputs of those encoding functions, effectively decomposing the constraints in the execution into two sets and ignoring the set of hard-to-solve constraints introduced by the encoding function.

The results of our encoding function identification, for the first iteration of the exploration, are summarized in Table 1, which presents on the left the program name and program size, the size of the input seed, and the number of instructions in the execution trace produced by the seed. The decryption and checksum columns describe the algorithm type and the maximum taint degree the algorithm produces in the execution. The rightmost column shows the runtime of the identification algorithm, which varies from a few seconds to close to two minutes. Because the identification is reused over a large number of iterations, the amortized overhead is even smaller.

Identifying the inverse functions. For Gheg and MegaD, BitFuzz needs to identify the inverse of the decryption function so that it can be used to re-stitch the inputs into a new program input for another iteration. (The encryption function for MegaD is the same one identified in previous work [4]; we use it to check the accuracy of our new identification approach.)

As described in Section 3.2.2, BitFuzz extracts the interface of each function in the execution trace that belongs to the same module as the decoding function, and then prioritizes them by the similarity of their interface to the decoding function. For both Gheg and MegaD, the function with the closest prototype is the encryption function, as our tool confirms by random testing with $k = 10$ tests. These samples illustrate the common pattern of a matching encryption function being included for two-way communication, so we did not need to search further afield for an inverse.

5.2 Decomposition vs. Non-Decomposition

In this section we compare the number of bugs found by BitFuzz when it uses decomposition and re-stitching, which we call *full* BitFuzz, and when it does not, which we call *vanilla* BitFuzz. Full BitFuzz uses the identified decoding functions to decompose the constraints into two sets, one with the constraints introduced by the decryption/checksum function and the other with the remaining constraints after that stage. In addition, each iteration of MegaD and Gheg uses the inverse function to re-stitch the inputs into a program input. Vanilla BitFuzz is comparable to previous dynamic symbolic execution tools. In both full and vanilla cases, BitFuzz detects bugs using the techniques described in Section 4.

In each iteration of its exploration, BitFuzz collects the execution trace of the malware program starting from the first time it receives network data. It stops the trace collection when the malware program sends back a reply, closes the communication socket, or a bug is detected. If none of those conditions is satisfied the trace collection is stopped after 2 minutes. For each collected trace, BitFuzz analyzes up to the first 200 input-dependent control flow branches and automatically generates new constraints that would explore new paths in the program. It then queries STP to solve each generated set of constraints, uses the solver's response to generate a new input, and adds it to the pool of inputs to test on future iterations. Because constraint solving can take a very long time without yielding a meaningful result, BitFuzz discards a set of constraints if STP runs out of memory or exceeds a 5-minute timeout for constraint solving.

We run both vanilla and full BitFuzz for 10 hours and report the bugs found, which are summarized in Table 2. Detailed descriptions of the bugs follow in Section 5.3. We break the results in Table 2 into three categories. The first category includes Zbot and MegaD for which full BitFuzz finds bugs but Vanilla BitFuzz does not. Full BitFuzz finds a total of 4 bugs, three in Zbot and

Name	Vulnerability type	Disclosure public identifier	Encoding functions	Search time (min.)	
				full	vanilla
Zbot	Null dereference	OSVDB-66499 [38]	checksum	17.8	>600
	Infinite loop	OSVDB-66500 [37]	checksum	129.2	>600
	Buffer overrun	OSVDB-66501 [36]	checksum	18.1	>600
MegaD	Process exit	n/a	decryption	8.5	>600
Gheg	Null dereference	OSVDB-66498 [35]	decryption	16.6	144.5
Cutwail	Buffer overrun	OSVDB-66497 [34]	none	39.4	39.4

Table 2: Description of the bugs our system finds in malware. The column "full" shows the results from the BitFuzz system including our decomposition and re-stitching techniques, while the "vanilla" column gives the results with these techniques disabled. ">600" means we run the tool for 10 hours and it is yet to find the bug.

one in MegaD. Three of the bugs are found in under 20 minutes and the second Zbot bug is found after 2 hours. Vanilla BitFuzz does not find any bugs in the 10-hour period. This happens due to the complexity of the constraints being introduced by the encoding functions. In particular, using full BitFuzz the 5-minute timeout for constraint solving is never reached and STP never runs out of memory, while using vanilla BitFuzz more than 90% of the generated constraints result in STP running out of memory.

The second category comprises Gheg for which both vanilla and full BitFuzz find the same bug. Although both tools find the same bug, we observe that vanilla BitFuzz requires almost ten times as long as full BitFuzz to do so. The cipher used by Gheg uses a one-byte hardcoded key that is combined with the first input byte using bitwise exclusive-or to produce the first output byte, that output byte is then used as key to encode the second byte also using bitwise exclusive-or and so on. Thus, the taint degree of the first output byte is one, for the second output byte is two and so on until the maximum taint degree of 128 shown in Table 1. The high maximum taint degree makes it harder for the solver to solve and explains why vanilla BitFuzz takes much longer than full BitFuzz to find the bug. Still, the constraints induced by the Gheg cipher are not as complex as the ones induced by the Zbot and MegaD ciphers and the solver eventually finds solutions for them. This case shows that even in cases where the solver will eventually find a solution, using decomposition and re-stitching can significantly improve the performance of the exploration.

The third category comprises Cutwail for which no encoding functions with high taint degree are identified and thus vanilla BitFuzz and full BitFuzz are equivalent.

In summary, full BitFuzz using decomposition and re-stitching clearly outperforms vanilla BitFuzz. Full BitFuzz finds bugs in cases where vanilla BitFuzz fails to do so due to the complexity of the constraints induced by the encoding functions. It also improves the performance of the exploration in other cases were the encoding constraints are not as complex and will eventually be solved.

5.3 Malware Vulnerabilities

In this section we present the results of our manual analysis to understand the bugs discovered by BitFuzz and our experiences reporting the bugs.

Zbot. BitFuzz finds three bugs in Zbot. The first one is a null pointer dereference. One of the C&C messages contains an array size field, which the program uses as the size parameter in a call to `RtlAllocateHeap`. When the array size field is larger than the available memory left in its local heap, the allocation returns a null pointer. The return value of the allocation is not checked by the program, which later attempts to write to the buffer, crashing when it tries to dereference the null pointer.

The second bug is an infinite loop condition. A C&C message comprises of a sequence of blocks. Each block has a 16-byte header and a payload. One of the fields in the header represents the size of the payload, s. When the trojan program finishes processing a block, it iteratively moves to the next one by adding the block size, $s + 16$, to a cursor pointer. When the value of the payload size is $s = -16$, the computed block size becomes zero, and the trojan keeps processing the same block over and over again.

The last bug is a stack buffer overrun. As mentioned above, a C&C message comprises of a sequence of blocks. One of the flags in the block header determines whether the block payload is compressed or not. If the payload is compressed, the trojan program decompresses it by storing the decompressed output into a fixed-size buffer located on the stack. When the length of the decompressed payload is larger than the buffer size, the program will write beyond the buffer. If the payload is large enough, it will overwrite a function return address and can eventually lead to control flow hijacking. This vulnerability is exploitable and we have successfully crafted a C&C message that exploits the vulnerability and hijacks the execution of the malware.

MegaD. BitFuzz finds one input that causes the MegaD bot to exit cleanly. We analyzed this behavior using the MegaD grammar produced by previous work [6] and found that the bug is present in the handling of the *ping* message (type `0x27`). If the bot receives a ping message and the bot identifier (usually set by a previously received C&C message) has not been set, then it sends a reply *pong* message (type `0x28`) and terminates. This behavior highlights the fact that, in addition to bugs, our stitched dynamic symbolic execution can also discover C&C messages that cause the malware to cleanly exit (e.g., kill commands), if those commands are available in the C&C protocol. These messages cannot be considered bugs but can still be used to disable the malware. They are specially interesting because they may have been designed to completely remove all traces of the malware running in the compromised host. In addition, their use could raise fewer ethical and legal questions than the use of an exploit would.

Gheg. BitFuzz finds one null pointer dereference bug in Gheg. The bug is similar to the one in Zbot. One of the C&C messages contains an array size field, whose value is multiplied by a constant (0x1e8) and the result used as the size parameter in a call to `RtlAllocateHeap`. The return value of the allocation is not checked by the program and the program later writes into the allocated buffer. When the array size field value is larger than the available memory in its local heap, the allocation fails and a null

Family	MD5	First seen	Reported by
Zbot	0bf2df85*7f65	Jun-23-09	Prevx
	1c9d16db*7fc8	Aug-17-09	Prevx
	7a4b9ceb*77d6	Dec-14-09	ThreatExpert
MegaD	700f9d28*0790	Feb-22-08	Prevx
	22a9c61c*e41e	Dec-13-08	Prevx
	d6d00d00*35db	Feb-03-10	VirusTotal
	09ef89ff*4959	Feb-24-10	VirusTotal
Gheg	287b835b*b5b8	Feb-06-08	Prevx
	edde4488*401e	Jul-17-08	Prevx
	83977366*b0b6	Aug-08-08	ThreatExpert
	cdbd8606*6604	Aug-22-08	Prevx
	f222e775*68c2	Nov-28-08	Prevx
Cutwail	1fb0dad6*1279	Aug-03-09	Prevx
	3b9c3d65*07de	Nov-05-09	Prevx

Table 3: Bug reproducibility across different malware variants. The shaded variants are the ones used for exploration.

pointer is returned. The program fails to check that the returned value is a null pointer and tries to dereference it.

Cutwail. BitFuzz finds a buffer overrun bug that leads to an out-of-bounds write in Cutwail. One of the received C&C messages contains an array. Each record in the array has a length field specifying the length of the record. This field is used as the size parameter in a call to `RtlAllocateHeap`. The returned pointer is appended to a global array that can only hold 50 records. If the array in the received message has more than 50 records, the 51^{st} record will be written outside the bounds of the global array. Near the global array, there exists a pointer to a private heap handle and the out-of-bounds write will overwrite this pointer. Further calls to `RtlAllocateHeap` will then attempt to access the malformed heap handle, and will lead to heap corruption and a crash.

Reporting the bugs. We reported the Gheg bug to the editors of the Common Vulnerabilities and Exposures (CVE) database [9]. Our suggestion was that vulnerabilities in malware should be treated similarly to vulnerabilities in commercial or open source programs, of course without reporting back to the developers. However, the CVE editors felt that malware vulnerabilities were outside the scope of their database. Subsequently, we reported the Gheg vulnerability to the Open Source Vulnerability Database (OSVDB) moderators who accepted it. Since then, we have reported all other vulnerabilities except the MegaD one, which may be considered intended functionality by the botmaster. Table 2 presents the public identifiers for the disclosed vulnerabilities. We further address the issue of disclosing malware vulnerabilities in Section 6.

5.4 Bug Persistence over Time

Bot binaries are updated very often to avoid detection by antivirus tools. One interesting question is how persistent over time are the bugs found by BitFuzz. To evaluate this, we retest our crashing inputs on other binaries from the same malware families. Table 3 shows all the variants, with the shaded variants corresponding to the ones explored by BitFuzz and mentioned in Table 1.

We replay the input that reproduces the bug BitFuzz found on the shaded variant on the rest of variants from the same family. As shown, the bugs are reproducible across all the variants we tested. These means for instance that the MegaD bug has been present for at least two years (the time frame covered by our variants). In addition, the MegaD encryption and decryption functions (and the key they use), as well as the C&C protocol have not changed, or barely evolved, through time. Otherwise the bug would not be reproducible in older variants. The results for Gheg are similar. The bug reproduces across all Gheg variants, although in this case our most recent sample is from November, 2008. Note that, even though the sample is relatively old it still works, meaning that it still connects to a C&C server on the Internet and sends spam. For Zbot, all three bugs reproduce across all variants, which means they have been present for at least 6 months. These results are important because they demonstrate that there are components in bot software, such as the encryption functions and C&C protocol grammar, that tend to evolve slowly over time and thus could be used to identify the family to which an unknown binary belongs, one widespread problem in malware analysis.

6. DISCUSSION

In light of our results, this section provides additional discussion on the applications for the discovered bugs and associated ethical considerations. Then, it presents a potential scenario for using the discovered bugs, and describes some limitations of our approach.

6.1 Applications and Ethical Considerations

Malware vulnerabilities could potentially be used in different “benign” applications such as remediating botnet infestations, for malware genealogy since we have shown that the bugs persist over long periods of time, as a capability for law enforcement agencies, or as a strategic resource in state-to-state cyberwarfare [39]. However, their use raises important ethical and legal questions. For example, there may be a danger of significant negative consequences, such as adverse effects to the infected machines. Also, it is unclear which legal entity would perform such remediation, and whether currently there exists any entity with the legal right to take such action. On the other hand, having a potential avenue for cleanup and not making use of it also raises some ethical concerns since if such remediation were effective, it would be a significant service to the malware’s future third-party victims (targets of DDoS attacks, spam recipients, etc.). Such questions belong to recent and ongoing discussions about ethics in security research (e.g., [13]) that have not reached a firm conclusion.

Malware vulnerabilities could also be used for malign purposes. For instance, there are already indications that attackers are taking advantage of known vulnerabilities in web interfaces used to administer botnets to hijack each others’ botnets [11]. This raises concerns about disclosing such bugs in malware. In the realm of vulnerabilities in benign software, there has been significant debate on what disclosure practices are socially optimal and there is a partial consensus in favor of some kind of “responsible disclosure” that gives authors a limited form of advance notice. However, it is not clear what the analogous best practice for malware vulnerabilities should be. We have faced this disclosure issue when deciding whether to publicly disclose the vulnerabilities we found and to which extent we should describe the vulnerabilities in the paper. We hope this paper strikes a fine balance but we also believe further discussion is needed on the proper avenue for disclosing malware vulnerabilities.

Potential application scenario. While we have not used our crashing inputs on bots in the wild, here we hypothetically discuss one possible scenario of how one might do so. The malware programs we analyze start TCP connections with a remote C&C server. To exploit the vulnerabilities we have presented, we need to impersonate the C&C server and feed inputs in the response to the initial request from the malware program. This scenario often happens during a botnet takedown, in which law enforcement or other responding entities identify the IP addresses and DNS names associated with the C&C servers used by a botnet, and appeal to relevant ISPs and registrars to have them de-registered or redirected to the responders. The responders can then impersonate the C&C server: one common choice is a *sinkhole server* that collects statistics on requests but does not reply. But such responders are also in a position to perform more active communication with bots, and for instance vulnerabilities like the ones we present could be used for cleanup if the botnet does not support cleanup via its normal protocol. For example, such a scenario happened recently during the attempted MegaD takedown by FireEye [29]. For a few days FireEye ran a sinkhole server that received the C&C connections from the bots. This sinkhole server was later handed to the Shadowserver Foundation [41].

6.2 Limitations

We have found our techniques to be quite effective against the current generation of malware. But since malware authors have freedom in how they design encoding functions, and an incentive to avoid analysis of their programs, it is valuable to consider what measures they might take against analysis.

Preventing access to inverses. To stitch complete inputs in the presence of a surjective transformation, our approach requires access to an appropriate inverse function: for instance, the encryption function corresponding to a decryption function. So far, we have been successful in finding such inverses either within the malware binary, or from standard sources, but these approaches could be thwarted if malware authors made different choices of cryptographic algorithms. For instance, malware authors could design their protocols using asymmetric (public-key) encryption and digital signatures. Since we would not have access to the private key used by the C&C server, we could not forge the signature in the messages sent to the bot. We could still use our decomposition and re-stitching approach to find bugs in malware, because the signature verification is a basically a free side-condition that can be ignored. However, we could only build a exploit for our modified bot, as other bots will verify the (incorrect) signature in the message and reject it. Currently, most malware do not use public-key cryptography, but that may change. In the realm of symmetric encryption, malware authors could deploy different non-standard algorithms for the server-to-bot and bot-to-server directions of communication: though not theoretically infeasible, the construction of an encryption implementation from a binary decryption implementation might be challenging to automate. For instance, Kolbitsch et al. [25] faced such a situation in recreating binary updates for the Pushdo trojan, which was feasible only because the decryption algorithm used was weak enough to be inverted by brute force for small plaintexts.

Obfuscating encoding functions. Malware authors could potentially keep our system from finding encoding functions in binaries by obfuscating them. General purpose packing is not an obstacle to our dynamic approach, but more targeted kinds of obfuscation would be a problem. For instance, our current implementation recognizes only standard function calls and returns, so if a malware author rewrote them using non-standard instructions our tool would require a corresponding generalization to compensate. Further along the arms race, there are also fundamental limitations arising from our use of a dynamic dependency analysis, similar to the limitations of dynamic taint analysis [8].

7. RELATED WORK

One closely related recent project is Wang et al.'s TaintScope system [43]. Our goals partially overlap with theirs in the area of checksums, but our work differs in three key aspects. First, Wang et al.'s techniques do not apply to decompression or decryption. Second, TaintScope performs exploration based on taint-directed fuzzing [17], while our system harnesses the full generality of symbolic execution. (Wang et al. use symbolic execution only for inverting the *encodings* of checksums, a task which is trivial in our applications.) Third, Wang et al. evaluate their tool only on benign software, while we perform the first automated study of vulnerabilities in malware.

The encoding functions we identify within a program can also be extracted from a program to be used elsewhere. The Binary Code Reuse [4] and Inspector Gadget [25] systems can be used to extract encryption and checksum functionalities, including some of the same ones our tool identifies, for applications such as network defense. Our application differs in that our system can simply execute the code in its original context instead of extracting it. Inspector Gadget [25] can also perform so-called gadget inversion, which is useful for the same reasons as we search for existing inverse functions. However, their approach does not work on strong cryptographic functions.

Previous work in protocol reverse engineering has used alternative heuristics to identify cryptographic operations in malware binaries. For instance ReFormat [44] proposes detecting such functions by measuring the ratio of arithmetic and bitwise instructions to other instructions. Our use of taint degree as a heuristic is more specifically motivated by the limitations of symbolic execution: for instance a simple stream cipher would be a target of the previous approaches but is not for this paper.

Decomposition is a broad class of techniques in program analysis and verification, but most previous decomposition techniques are symmetric in the sense that each of the sub-components of the program are analyzed similarly, while a key aspect of our approach is that different components are analyzed differently. In analysis and verification, decomposition at the level of functions, as in systems like Saturn [45], is often called a compositional approach. In the context of tools based on symbolic execution, Godefroid [18] proposes a compositional approach that performs dynamic symbolic execution separately on each function in a program. Because this is a symmetric technique, it would not address our problem of encoding functions too complex to analyze even in isolation. More similar to our approach is grammar-based fuzzing [5, 19], an instance of serial decomposition. However parsers require different specialized techniques than encoding functions.

8. CONCLUSION

We have presented a new approach, stitched dynamic symbolic execution, to allow analysis in the presence of functionality that would otherwise be difficult to analyze. Our techniques for automated identification, decomposition, and re-stitching allow our system to bypass functions like decryption and checksum verification to find bugs in core program logic. Specifically, these techniques enable the first automated study of vulnerabilities in malware. Our BitFuzz tool finds 6 unique bugs in 4 prevalent malware families. These bugs can be triggered over the network to terminate or take control of a malware instance. These bugs have persisted across malware revisions for months, and even years. There are still many unanswered questions about the applications and ethical concerns surrounding malware vulnerabilities, but our results demonstrate that vulnerabilities in malware are an important security resource that should be the focus of more research in the future.

9. ACKNOWLEDGMENTS

We would like to specially thank Edward Xuejun Wu for his proof of concept exploit for the Zbot buffer overrun vulnerability and for his help in the Cutwail bot experiment. We also thank Chia Yuan Cho for his help understanding the MegaD process exit bug and the anonymous reviewers for their insightful comments. This work was done while Juan Caballero and Pongsin Poosankam were visiting student researchers at University of California, Berkeley.

10. REFERENCES

[1] BitBlaze: Binary analysis for computer security. `http://bitblaze.cs.berkeley.edu/`.

[2] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of the 16th USENIX Security Symposium*, pages 213–228, Montreal, Quebec, Canada, Aug. 2007.

[3] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *IEEE Symposium on Security and Privacy*, 2008.

[4] J. Caballero, N. M. Johnson, S. McCamant, and D. Song. Binary code extraction and interface identification for security applications. In *NDSS'10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*, pages 391–408, San Diego, California, USA, Mar. 2010.

[5] J. Caballero, Z. Liang, P. Poosankam, and D. Song. Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration. In *RAID'09: Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, Saint-Malo, France, Sept. 2009.

[6] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *CCS'09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 621–634, Chicago, Illinois, USA, Nov. 2009.

[7] C. Cadar and D. R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN'05: Proceedings of the 12th International SPIN Workshop on Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 2–23, San Francisco, California, USA, Aug. 2005.

[8] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA'08: Proceedings of the Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, pages 143–163, Paris, France, July 2008.

[9] CVE: Common vulnerabilities and exposures. `http://cve.mitre.org/`.

[10] J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, Heidelberg, Germany, Mar. 2002.

[11] D. Danchev. Help! someone hijacked my 100k+ Zeus botnet!, Feb. 2009. `http://ddanchev.blogspot.com/2009/02/help-someone-hijacked-my-100k-zeus.html`.

[12] D. De, A. Kumarasubramanian, and R. Venkatesan. Inversion attacks on secure hash functions using SAT solvers. In *SAT'07: Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing*, volume 4501 of *Lecture Notes in Computer Science*, pages 377–382, Lisbon, Portugal, 2007.

[13] D. Dittrich, F. Leder, and T. Werner. A case study in ethical decision making regarding remote mitigation of botnets. In *WECSR'10: Workshop on Ethics in Computer Security Research*, Lecture Notes in Computer Science, Tenerife, Canary Islands, Spain, Jan. 2010.

[14] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Trans. Software Eng.*, 10(4):438–444, 1984.

[15] Security guru gives hackers a taste of their own medicine, Apr. 2008. `http://www.wired.com/threatlevel/2008/04/researcher-demo`.

[16] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV'07: Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 519–531, Berlin, Germany, July 2007.

[17] V. Ganesh, T. Leek, and M. C. Rinard. Taint-based directed whitebox fuzzing. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, pages 474–484, Vancouver, British Columbia, Canada, May 2009.

[18] P. Godefroid. Compositional dynamic test generation. In *POPL'07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, Nice, France, Jan. 2007.

[19] P. Godefroid, A. Kieżun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *PLDI'08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 206–215, Tucson, Arizona, USA, June 2008.

[20] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI'05: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, Illinois, USA, June 2005.

[21] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS'08: Proceedings of the Network and Distributed System Security Symposium*, San Diego, California, USA, Feb. 2008.

[22] M. Kassner. The top 10 spam botnets: New and improved, Feb. 2010. `http://blogs.techrepublic.com.com/10things/?p=1373`.

[23] A. Kieżun, P. J. Guo, K. Jayaraman, and M. D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *ICSE'09: Proceedings of the 31st International Conference on Software Engineering*, pages 199–209, Vancouver, British Columbia, Canada, May 2009.

[24] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[25] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector Gadget: Automated extraction of proprietary gadgets from malware binaries. In *SP'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2010.

[26] J. Leyden. Monster botnet held 800,000 people's details. *The Register*, Mar. 2010. `http://www.theregister.co.uk/2010/03/04/mariposa_police_hunt_more_botherders/`.

[27] M86 Security Labs. Botnet statistics for week ending April 11, 2010, Apr. 2010. `http://www.m86security.com/labs/bot_statistics.asp`.

[28] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI'08: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, pages 193–205, Tucson, Arizona, USA, June 2008.

[29] Smashing the Mega-d/Ozdok botnet in 24 hours. `http://blog.fireeye.com/research/2009/11/smashing-the-ozdok.html`.

[30] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[31] D. Molnar, X. C. Li, and D. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th USENIX Security Symposium*, pages 67–81, Montreal, Quebec, Canada, Aug. 2009.

[32] National Institute of Standards and Technology, Gaithersburg, MD, USA. *Federal Information Processing Standard 180-2: Secure Hash Standard*, Aug. 2002.

[33] OpenSSL: The open source toolkit for SSL/TLS. `http://www.openssl.org/`.

[34] OSVDB. Cutwail Bot svchost.exe CC Message Handling Remote Overflow, July 2010. `http://osvdb.org/66497`.

[35] OSVDB. Gheg Bot RtlAllocateHeap Function Null Dereference Remote DoS, July 2010. `http://osvdb.org/66498`.

[36] OSVDB. Zbot Trojan svchost.exe Compressed Input Handling Remote Overflow, July 2010. `http://osvdb.org/66501`.

[37] OSVDB. Zbot Trojan svchost.exe Network Message Crafted Payload Size Handling Infinite Loop Remote DoS, July 2010. `http://osvdb.org/66500`.

[38] OSVDB. Zbot Trojan svchost.exe RtlAllocateHeap Function Null Dereference Remote DoS, July 2010. `http://osvdb.org/66499`.

[39] W. A. Owens, K. W. Dam, and H. S. Lin, editors. *Technology, Policy, Law, and Ethics Regarding U.S. Acquisition and Use of Cyberattack Capabilities*. The National Academies Press, Washington, DC, USA, 2009.

[40] B. Potter, Beetle, CowboyM, D. Moniz, R. Thayer, 3ricj, and Pablos. Shmoo-fu: Hacker goo, goofs, and gear with the shmoo. In *DEFCON*, Las Vegas, Nevada, USA, July 2005. http://www.defcon.org/images/defcon-13/dc13-presentations/dc-13-beetle-shmoo-fu.pdf.

[41] Shadowserver foundation. `http://www.shadowserver.org/`.

[42] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis (keynote invited paper). In *ICISS'08: Proceedings of the 4th International Conference on Information Systems Security*, volume 5352 of *Lecture Notes in Computer Science*, pages 1–25, Hyderabad, India, Dec. 2008.

[43] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *SP'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2010.

[44] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. ReFormat: Automatic reverse engineering of encrypted messages. In *ESORICS'09: 14th European Symposium on Research in Computer Security*, volume 5789 of *Lecture Notes in Computer Science*, pages 200–215, Saint-Malo, France, Sept. 2009.

[45] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 351–363, Long Beach, California, USA, Jan. 2005.

[46] The zlib library. `http://www.zlib.net/`.