X-Force: Force-Executing Binary Programs for Security Applications

Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu
Purdue University
{*pengf, deng14, xyzhang, dxu*}*@cs.purdue.edu*

Zhiqiang Lin
UT Dallas
zhiqiang.lin@utdallas.edu

Zhendong Su
UC Davis
su@cs.ucdavis.edu

Abstract

This paper introduces X-Force, a novel binary analysis engine. Given a potentially malicious binary executable, X-Force can force the binary to execute requiring no inputs or proper environment. It also explores different execution paths inside the binary by systematically forcing the branch outcomes of a very small set of conditional control transfer instructions. X-Force features a crash-free execution model that can detect and recover from exceptions. In particular, it can fix invalid memory accesses by allocating memory on-demand and setting the offending pointers to the allocated memory. We have applied X-Force to three security applications. The first is to construct control flow graphs and call graphs for stripped binaries. The second is to expose hidden behaviors of malware, including packed and obfuscated APT malware. X-Force is able to reveal hidden malicious behaviors that had been missed by manual inspection. In the third application, X-Force substantially improves analysis coverage in dynamic type reconstruction for stripped binaries.

1 Introduction

Binary analysis has many security applications. For example, given an unknown, potentially malicious executable, binary analysis helps construct its human inspectable representations such as control flow graph (CFG) and call graph (CG), with which security analysts can study its behavior [40, 23, 50, 46, 6, 33]. Binary analysis also helps identify and patch security vulnerabilities in COTS binaries [10, 14, 31, 51, 11]. Valuable information can be reverse-engineered from executables through binary analyses. Such information includes network protocols [44, 12, 7, 47, 28, 32], input formats [27, 29, 13], variable types, and data structure definitions [30, 25, 39]. They can support network sniffing, exploit generation, VM introspection, and forensic analysis.

Existing binary analysis can be roughly classified into static, dynamic, and symbolic (concolic) analysis. Static analysis analyzes an executable directly without executing it; dynamic analysis acquires analysis results by executing the subject binary; symbolic (concolic) analysis is able to generate inputs to explore different paths of a binary. These different styles of analyses have their respective strengths and limitations. Static analysis has difficulty in handling packed and obfuscated binaries. Memory disambiguation and indirect jump/call target analysis are known to be very challenging for static analysis.

Dynamic binary analysis is based on executing the binary on a set of inputs. It is widely used in analyzing malware. However, dynamic analysis is incomplete by nature. The quality of analysis results heavily relies on coverage of the test inputs. Moreover, modern malware [16, 26, 19] has become highly sophisticated, posing many new challenges for binary analysis: (1) For a zero-day binary malware, we typically do not have any knowledge about it, especially the nature of its input, making traditional execution-based analysis [15, 50, 4, 43, 49] difficult; (2) Malware binaries are increasingly equipped with anti-analysis logic [37, 5, 17, 18, 35] and hence may refuse to run even if given valid input; (3) Malware binaries may contain multi-staged, condition-guarded, and environment-specific malicious payloads, making it difficult to reveal all payloads, even if one manages to execute them.

Symbolic [8] and concolic analysis [38, 20, 40, 10] has seen much progress in recent years. Some handle binary programs [40, 10, 33, 6] and can explore various paths in a binary. However, difficulties exist when scaling them to complex, real-world binaries, as they operate by modeling individual instructions as symbolic constraints and using SMT/SAT solvers to resolve the generated constraints. Despite recent impressive progress, SMT/SAT remains expensive. While symbolic and concrete executions can be performed simultaneously so that concrete execution may help when symbolic analysis encounters difficulties, the user needs to provide concrete inputs, called *seed inputs*, and the quality of seed inputs is critical to the execution paths that can be explored. With no or little knowledge about malware input, creating such seed inputs is difficult. Moreover, many existing techniques cannot handle obfuscated or self-modifying binaries.

In this paper, we propose a new, practical execution engine called X-Force. The core enabling technique behind X-Force is *forced execution* which, as its name suggests, *forces an arbitrary binary to execute along different paths without any input or environment setup.* More specifically, X-Force monitors the execution of a binary through dynamic binary instrumentation, systematically forcing a small set of instructions that may affect the execution path (e.g., predicates and jump table ac-

cesses) to have specific values, regardless of their computed values, and supplying random values when inputs are needed. As such, the *concrete* program state of the binary can be systematically explored. For instance, a packed/obfuscated malware can be forced to unpack/deobfuscate itself by setting the branch outcomes of self-protection checks, which terminate execution in the presence of debugger or virtual machine. X-Force is able to tolerate invalid memory accesses by performing on-demand memory allocations. Furthermore, by exploring the reachable state of a binary, X-Force is able to explore different aspects or stages of the binary behavior. For example, we can expose malware's data exfiltration operations, without the presence of the real data asset being targeted.

Compared to manual inspection and static analysis, X-Force is more accurate as many difficulties for static analysis, such as handling indirect jumps/calls and obfuscated/packed code, can be substantially mitigated by the concrete execution of X-Force. Compared to symbolic/concolic analysis, X-Force trades precision slightly for *practicality* and *extensibility*. Note that X-Force may explore infeasible paths as it forces predicate outcomes; whereas symbolic analysis attempts to respect path feasibility through constraint solving[1]. The essence of X-Force will be discussed later in Section 6. Furthermore, executions in X-Force are all concrete. Without the need for modeling and solving constraints, X-Force is more likely to scale to large programs and long executions. The concrete execution of X-Force makes it suitable for analyzing packed and obfuscated binaries. It also makes it easy to port existing dynamic analysis to X-Force to leverage the large number of executions, which will mitigate the incompleteness of dynamic analyses.

Our main contributions are summarized as follows:

- We propose X-Force, a system that can force a binary to execute requiring no inputs or any environment setup.
- We develop a crash-free execution model that could detect and recover from exceptions properly. We have also developed various execution path exploration algorithms.
- We have overcome a large number of technical challenges in making the technique work on real world binaries including packed and obfuscated malware binaries.
- We have developed three applications of X-Force. The first is to construct CFG and CG of stripped binaries, featuring high quality indirect jump and call target identification; the second is to study hidden behavior of advanced malwares; the third one is to apply X-Force in reverse engineering variable types and data structure definitions of executables. Our results show that X-Force substantially advances the state-of-the-arts.

[1]However, due to the difficulty of precisely modeling program behavior, even state-of-the-art symbolic analysis techniques [8, 10, 40] cannot guarantee soundness.

2 Motivation Example

Consider the snippet in Figure 1. It shows a hidden malicious payload that hijacks the name resolution for a specific domain (line 14), which varies according to the current date (in function *genName()*). In particular, it receives some integer input at line 2. If the input satisfies condition *C* at line 3, a *DNSentry* object will be allocated. In lines 5-8, if the input has the *CODE_RED* bit set, it populates the object by calling *genName()* and stores the input and the generated name as a (key, value) pair into a hash table. In lines 12-14, the pair is retrieved and used to guide domain name redirection. Note that the hash table is used as a general storage for objects of various types. In line 10, an irrelevant object *o* is also inserted into the table.

This example illustrates some of the challenges faced by both static and symbolic/concolic analysis. *In static analysis*, it is difficult to determine that the object retrieved at line 12 is the one inserted at line 7 because the abstract domain has to precisely model the behavior of the hash table put/get operations and the condition that `y==x`, which requires context-sensitive and path-sensitive analysis, and disambiguating the memory `bucket[i]` and `bucket[i+4]` in *table_get()* and *table_put()*. The approximations made by many static analysis techniques often determine the object at line 12 could be the one put at line 7 or 10. Performed solely at the binary level, such an analysis is actually much more challenging than described here. *In symbolic/concolic analysis*, one can model the input at line 2 as a symbolic variable such that, by solving the symbolic constraints corresponding to path conditions, the hidden payload might be reached. However, the dictionary read at line 21 will be difficult to handle if the file is *unavailable*. Modeling the file as symbolic often causes scalability issues if it has nontrivial format and size, because the generated symbolic constraints are often complex and the search space for acquiring syntactically correct inputs may be extremely large.

In X-Force, the binary is first executed as usual by providing random inputs. Note that X-Force does not need to know the input format *a priori* as its exception recovery mechanism prevents any crashes/exceptions. In other words, the supply of random input values is merely to allow the execution to proceed, not to drive the execution along different paths. In the first normal run, assume that the false branches of the conditionals at lines 3, 5 and 13 are taken, yielding an uninteresting execu-

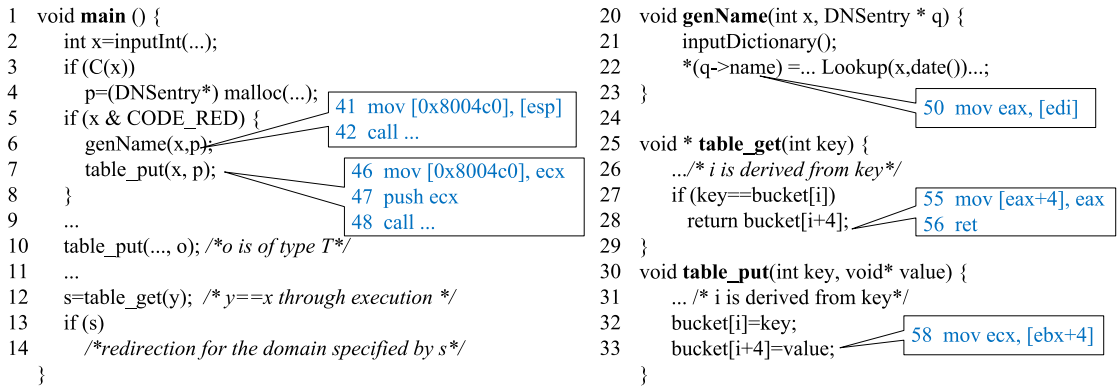

Figure 1: Motivating Example.

tion. X-Force will then try to *force-set* branch outcomes at a small number (say, 1 or 2) of predicates by performing systematic search. Assume that the branch outcome at line 5 is force-set to "true". The malicious payload will be forced to activate. Note that pointer p has a null value at line 6, which will normally crash the execution at line 22. X-Force tolerates such invalid accesses by *allocating memory on demand*, right before line 22. Also, even if the dictionary file at line 21 is absent, X-Force will force it through by supplying random input values. As such, some random integer and domain are inserted into the table (line 7) and retrieved later (line 12). Eventually, the random domain name is redirected at line 14, exposing the DNS hijacking operation. We argue that the domain name itself is not important as long as the hidden hijacking logic is exposed.

3 High Level Design

3.1 Forced Execution Semantics

This section explains the basics of how a single forced execution proceeds. The goal is to have a non-crashable execution. For readability, we focus on explaining how to detect and recover from memory errors in this subsection, and then gradually introduce the other aspects of forced execution such as path exploration and handling libraries and threads in later sections.

Program	P	$::= s$
Stmt	s	$::= s_1; s_2 \mid \texttt{nop} \mid r :=^{\ell} e \mid r :=^{\ell} \texttt{R}(r_a) \mid \texttt{W}^{\ell}(r_a, r_v) \mid \texttt{jmp}^{\ell}(\ell_1) \mid \texttt{if}\ (r^{\ell})\ \texttt{then jmp}(\ell_1) \mid \texttt{jmp}^{\ell}(r) \mid r := \texttt{malloc}^{\ell}(r_s) \mid \texttt{free}^{\ell}(r) \mid \texttt{call}^{\ell}(\ell_1) \mid \texttt{call}^{\ell}(r) \mid \texttt{ret}^{\ell}$
Operator	op	$::= + \mid - \mid * \mid / \mid > \mid < \mid \ldots$
Expr	e	$::= c \mid a \mid r_1\ \texttt{op}\ r_2$
Register	r	$::= \{esp, eax, ebx, \ldots\}$
Const	c	$::= \{true, false, 0, 1, 2, \ldots\}$
Addr	a	$::= \{0, MIN\_ADDR, MIN\_ADDR+1, \ldots, MAX\_ADDR\}$
PC	ℓ	$::= \{\ell_1, \ell_2, \ell_3, \ldots\}$

Figure 2: Language.

Language. Due to the complexity of the x86 instruction set, we introduce a simple low-level language that models x86 binary executables to facilitate discussion. We only model a subset that is sufficient to illustrate the key ideas. Fig. 2 shows the syntax.

Memory reads and writes are modeled by $\texttt{R}(r_a)$ and $\texttt{W}(r_a, r_v)$ with r_a holding the address and r_v the value. Since it is a low-level language, we do not model conditional or loop statements, but rather guarded jumps; `malloc()` and `free()` represent heap allocation and deallocation. Function invocations and returns are modeled by `call()` and `ret`. In our language, stack/heap memory addresses are modeled as a range of integers and a special value 0 to denote the null pointer value. Program counters (or instruction addresses) are explicitly modeled by the *PC* set. Observe that each instruction is labeled with a *PC*, denoting its instruction address. Direct jumps/calls are parameterized with explicit *PC* values whereas indirect jumps/calls are parameterized with a register.

LSet		$::= \mathcal{P}(Addr)$
$SR \in RegLinearSet$		$::= Register \mapsto \&LSet$
$SM \in MemLinearSet$		$::= Addr \mapsto \&LSet$
$accessible \in AddrAccessible$		$::= Addr \mapsto boolean$

recovery (r) ::=

1: $S \leftarrow SM(r)$
2: $VS \leftarrow \{\}$
3: **for** each address $a \in S$ **do**
4: $\quad VS \leftarrow VS + \{*(a)\}$
5: **end for**
6: $min \leftarrow$ the minimal value in VS
7: $max \leftarrow$ the maximum value in VS
8: $t \leftarrow malloc(max - min + BLOCKSIZE)$
9: $accessible[t, t + max - min + BLOCKSIZE - 1] = true$
10: **for** each address $a \in S$ **do**
11: $\quad offset \leftarrow *(a) - min$
12: $\quad *(a) \leftarrow t + offset$
13: **end for**

Figure 3: Definitions.

In X-Force, we ensure that an execution is not crash-

Table 1: Linear Set Computation Rules.

Statement	Action[1,2]	Rule
initially	foreach (global address t) if ($isAddr(*t)$) $SM(t) = \{t\}$;	L-INIT
$r := \mathtt{R}(r_a)$	$SR(\text{“}r\text{”}) \rightarrow nil$; if($SM(r_a)$) $SR(\text{“}r\text{”}) \rightarrow SM(r_a)$;	L-READ

$\mathtt{W}(r_a, r_v)$	if ($SM(r_a)$) $SM(r_a) = SM(r_a) - \{r_a\}$
	$SM(r_a) \rightarrow nil$;
	if ($SR(\text{“}r_v\text{”})$)
	  $SR(\text{“}r_v\text{”}) = SR(\text{“}r_v\text{”}) \cup \{r_a\}$;
	  $SM(r_a) \rightarrow SR(\text{“}r_v\text{”})$;
		L-WRITE

$r := a$	$SR(\text{“}r\text{”}) \rightarrow \{\}$	L-ADDR
$r := c$ /*!$isAddr(c)$*/	$SR(\text{“}r\text{”}) \rightarrow nil$	L-CONST

$r := r_1 +/- r_2$	if (!($isAddr(r_1)$&&$isAddr(r_2)$))
	  $SR(\text{“}r\text{”}) \rightarrow nil$
	  if ($isAddr(r_1)$) $SR(\text{“}r\text{”}) \rightarrow SR(\text{“}r_1\text{”})$;
	  if ($isAddr(r_2)$) $SR(\text{“}r\text{”}) \rightarrow SR(\text{“}r_2\text{”})$;
		L-LINEAR

$r := r_1 *\!/ \ldots r_2$	$SR(\text{“}r\text{”}) \rightarrow nil$	L-NON-LNR

free(r)	$t = r$;
	while ($accessible(t)$)
	  if ($SM(t)$) $SM(t) = SM(t) - \{t\}$;
	  $t++$;
		L-FREE

1. The occurrence“r” denotes the symbolic name of register r, the occurrence of r denotes the value stored in r.
2. Operator “=” means set update, “$\rightarrow$” means pointer update.

able by allocating memory on-demand. However, when we replace a pointer pointing to an invalid address a with the allocated memory, *we need to update all the other pointer variables that have the same address value or a value denoting an offset from the address*. We achieve this through the *linear set tracing semantics*, which is also the basic semantics for forced executions[2]. Its goal is to identify the set of variables (i.e. memory locations and registers at the binary level), whose values have *linear correlations*. In this paper, *we say two variables are linearly correlated if the value of one variable is computed from the value of the other variable by adding or subtracting a value.* Note that it is simpler than the traditional definition of linear correlation, which also allows a scaling multiplier. It is however sufficient in this work as *the goal of linear set tracing is to identify correlated pointer variables, which are induced by address offsettings that are exclusively additions and subtractions*.

The semantics is presented in Table 1. The corresponding definitions are presented in Fig 3. Particularly, linear set *LSet* denotes a set of addresses such that *the values stored in these addresses are linearly correlated*. Mapping *SR* maps a register to the reference of a *LSet*. Intuitively, one could interpret that it maps a register to a pointer pointing to a set of addresses such that the values stored in the register and those addresses are linearly correlated. Two registers map to the same reference (of a *LSet*) implies that the values of the two registers are also linearly correlated. Similarly, mapping *SM* maps an address to the reference of a *LSet* such that the values in the address and all the addresses in *LSet* are linearly correlated. The essence of linear set tracing is to maintain the *SR* and *SM* mappings for all registers and addresses that have been accessed so that at any execution point, we can query the set of linearly correlated variables of any given variable.

[2]We will explain the predicate switching part of the semantics in Section 3.2

Table 2: Memory Error Prevention and Recovery.

Statement	Action	Rule
$r :=$ malloc(r_1)	for ($i = r$ to $r + r_1 - 1$) $accessible(i) = true$	M-ALLOC

free(r)	$t = r$;
	while ($accessible(t)$)
	  $accessible(t) = false$
	  $t++$;
		M-FREE

$r := \mathtt{R}(r_a)$	if (!$accessible(r_a)$) $recovery(r_a)$;	M-READ
$\mathtt{W}(r_a, r_v)$	if (!$accessible(r_a)$) $recovery(r_a)$;	M-WRITE

Before execution, the *SM* mapping of all global variables that have an address value is set to the address itself, meaning the variable is only linearly correlated with itself initially (rule L-INIT). Function *isAddr(v)* determines if a value v could be an address. X-Force monitors all memory allocations and the image loading process. Thus, given a value, X-Force treats it as a pointer if it falls into static, heap, or stack memory regions. Note that we do not need to be sure that the value is indeed an address. Over-approximations only cause some additional linear set tracing. For a memory read operation, the *SR* mapping of the destination register points to the *SM* set of the value in the address register if the *SM* set exists, which implies the value is an address, otherwise it is set to nil (rule L-READ). Note that in the rule we use “r” to denote the symbolic name of r and r_a to denote the value stored in r_a. $SR(\text{“}r\text{”}) \rightarrow SM(r_a)$ means that we set $SR(\text{“}r\text{”})$ to point to the $SM(r_a)$ set. For a memory write, we first eliminate the destination address from its linear set. Then, the address is added to the linear set of the value register as the address essentially denotes a new linearly correlated variable. Finally, the *SM* mapping of the address is updated (rule L-WRITE). Note that operation “=” means set update, which is different from “$\rightarrow$” meaning set reference update. For a simple address assignment, the *SR* set is set to pointing to an empty linear set, which is different from a nil value (rule L-ADDR). The empty set is essentially an *LSet* object that could be pointed to by multiple registers to denote their linear correlation. A nil value cannot serve this purpose. For a linear operator, the *SR* mapping of the destination register is set to pointing to the *SR* mapping of the register holding an address value (rule L-LINEAR). Intuitively, this is because we are only interested in the linear correlation between address values (for the purpose of memory error recovery). For heap de-allocation, we have to remove each de-allocated address from its linear set (rule L-FREE).

Table 2 presents the set of memory error detection and

recovery rules. The relevant definitions are in Fig. 3. An auxiliary mapping $accessible()$ is introduced to denote if an address has been allocated and hence accessible. The M-ALLOC and M-FREE rules are standard. Upon reading or writing an un-accessible address, X-Force calls function *recovery ()* with the register holding the invalid address to perform recovery. In the function, we first acquire the values of all the variables in the linear set and identify the minimal and maximum values (lines 1-6). Note that the values may be different (through address offsetting operations). We then allocate a piece of memory on demand according to the range of the values and a pre-defined default memory block size. Then in lines 9-12, the variables in the linear set are updated according to their offsets in the block. We want to point out that on-demand allocation may not allocate enough space. However, such insufficiency will be detected when out-of-bound accesses occur and further on-demand re-allocation will be performed. We also want to point out that a correctly developed program would first write to an address before it reads. As such, the on-demand allocation is often triggered by the first write to an invalid buffer such that the value could be correctly written and later read. In other words, we do not need to recover values in the on-demand allocated buffers.

In our real implementation, we also update all the registers that are linearly correlated, which can be determined by identifying the registers pointing to the same set. Furthermore, the rules only describe how we ensure heap memory safety whereas X-Force protects critical stack addresses such as return addresses and parameters, which we will discuss later.

Example. Fig. 4 presents part of a sample execution with the linear set tracing and memory safety semantics. The program is from the motivation example (Fig. 1). In the execution, the else branch of line 3 is taken but the true branch of line 5 is forced. As such, pointer p has a null value when it is passed to function *genName()*, which would cause an exception at line 22. In Fig. 4, we focus on the executions of lines 6, 22 and 7. The second column shows the binary code (in our simplified language). The third column shows the corresponding linear set computation and memory exception detection and recovery. Initially, $SM(\&p = 0x8004c0)$ is set to pointing to the set $\{0x8004c0\}$ according to rule L-INIT. At binary code line 2, $SR(eax)$ is set to pointing to the set of $SM(\&p)$. At line 3, since the value is further copied to a stack address $0xce0080$, *eax*, $\&p$ and the stack address all point to the same linear set containing $\&p$ and the stack address. Intuitively, these are the three variables that are linearly correlated. At lines 9 and 10, *edi* further points to the same linear set. At line 12, when the program tries to access the address denoted by $edi = 4$, the memory safety component detects the exception and performs on demand allocation. According to the linear set, $\&p$ and the stack address $0xce0080$ are set to the newly allocated address $0xd34780$ while *edi* is updated to $0xd34784$ according to its offset. While it is not presented in the table, the program further initializes the newly allocated data structure. As a result, when pointer p is later passed to *table_put()*, it points to a valid data structure. □

Algorithm 1 Path Exploration Algorithm

Output:	Ex - the set of executions (each denoted by a sequence of switched predicates) achieving a certain given goal (e.g. maximum coverage)
Definition	*switches*: the set of switched predicates in a forced execution, denoted by a sequence of integers. For example, $1 \cdot 3 \cdot 5$ means that the 1st, 3rd, and 5th predicates are switched $WL : \mathcal{P}(\overline{Int})$ - a set of forced executions, each denoted by a sequence of switched predicates $preds : \overline{Predicate \times boolean}$ - the sequence of executed predicates with their branch outcomes

1: $WL \leftarrow \{\text{nil}\}$
2: $Ex \leftarrow$ nil
3: **while** WL **do**
4: $\quad switches \leftarrow WL$.pop()
5: $\quad Ex \leftarrow Ex \cup switches$
6: $\quad$ Execute the program and switch branch outcomes according to *switches*, update fitness function $\mathcal{F}$
7: $\quad preds \leftarrow$ the sequence of executed predicates
8: $\quad t \leftarrow$ the last integer in *switches*
9: $\quad preds \leftarrow$ remove the first t elements in *preds*
10: $\quad$ **for** each $(p,b) \in preds$ **do**
11: $\quad\quad$ **if** $eval(\mathcal{F},p,b)$ **then**
12: $\quad\quad\quad$ update fitness function $\mathcal{F}$
13: $\quad\quad\quad WL \leftarrow WL \cup switches \cdot t$
14: $\quad\quad$ **end if**
15: $\quad\quad t \leftarrow t+1$
16: $\quad$ **end for**
17: **end while**

In the early stage of the project, we tried a much simpler strategy that is to terminate a forced execution when an exception is observed. However, we observed that since we do not provide any real inputs, exceptions are very common. Furthermore, simply skipping instructions that cause exceptions did not work either because that would have cascading effects on program state corruption. Finally, a crash-proof execution model as proposed turned out to be the most effective one.

X-Force also automatically recovers from other exceptions such as division-by-zero, by skipping those instructions that cause exceptions. Details are omitted.

Source Code Trace	Binary Code Trace	Linear Set Computation and Memory Safety
6 genName(x,p);	1. ebx= 0x8004c0;	**SR(ebx)**→ {}
	2. eax= R(ebx); /* eax=0 */	**SR(eax)**, SM(0x8004c0) → {0x8004c0}
	3. W(esp, eax); /* esp=0xce0080 */	**SM(0xce0080)**, SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	4. …	
	5. call genName;	
/* in ***genName***(... DNSentry * q) */	6. ...	
22 *(q->name) =... Lookup(...)	7. ebx= ebp;	
	8. ebx= ebx + 8; /* ebx=0xce0080*/	
	9. edi= R(ebx);	**SR(edi)**, SM(0xce0080), SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	10. edi =edi + 4; /* edi= 0+4=4*/	**SR(edi)**, SM(0xce0080), SR(eax), SM(0x8004c0)→ {0x8004c0, 0xce0080}
	11. eax=…; /* eax=Lookup(...) */	SR(edi),SM(0xce0080),SM(0x8004c0)→ {0x8004c0, 0xce0080}
	12. W(edi, eax);	Exception! *0xce0080 = *0x8004c0 = malloc (4+BLOCKSIZE) = 0xd34780 edi= 0xd34780 + 4=0xd34784
7 table_put(x,p);	13. …	
	14. ebx= 0x8004c0;	
	15. ecx= R(ebx);	ecx= 0xd34780
	16. W(esp, ecx); /* esp=0xce0080 */	
	17. call table_put;	

Figure 4: Sample Execution for Linear Set Tracing and Memory Safety. The code is from Fig. 1.

3.2 Path Exploration in X-Force

One important functionality of X-Force is the capability of exploring different execution paths of a given binary to expose its behavior and acquire complete analysis results. In this subsection, we explain the path exploration algorithm and strategies.

To simplify discussion, we first assume a binary only performs control transfer through simple predicates (i.e. predicates with constant control transfer targets). We will introduce how to extend the algorithms in realistic settings, e.g., supporting exploration of indirect jumps in later section.

Algorithm 1 describes a general path exploration algorithm, which generates a pool of forced executions that are supposed to meet our goal specified by a configurable fitness function. It is a work list algorithm. The work list stores a list of (forced) executions that may be further explored by switching more predicates. Each execution is denoted by a sequence of integer numbers that specify the executed predicate instances to switch. Note that X-Force only force-sets the branch outcome of a small set of predicate instances. It lets the other predicate instances run as usual. Initially (line 1), the work list is a singleton set with a nil sequence, representing an execution without switching any predicate. Note that the work list is not empty initially. At the end of a forced execution, we update the fitness function that indicates the remaining space to explore (line 6), e.g., coverage. Then in lines 7-16, we try to determine if it would be of interest to further switch more predicate instances. Lines 7-9 compute the sequence of predicate instances eligible for switching. Note that it cannot be a predicate before the last switched predicate specified in *switches* as switching such a predicate may change the control flow such that the specification in *switches* becomes invalid. In lines 10-16, for each eligible predicate and its current branch outcome, we query the fitness function to determine if we should further switch it to generate a new forced execution. If so, we add it to the work list and update the fitness function. Note that in each new forced execution, we essentially switch one more predicate.

Different Fitness Functions. The search space of all possible paths is usually prohibitively large for real-world binaries. Different applications may define different fitness functions to control the scope they want to explore. In the following, we introduce three fitness functions that we use. Other more complex functions can be similarly developed.

- *Linear Search.* In certain applications, such as constructing control flow graphs and dynamic type reverse engineering (Section 5), the goal may be just to cover each instruction. The fitness function $\mathcal{F}$ could be defined as a mapping $covered : Predicate \times boolean \mapsto boolean$ that determines if a branch of a predicate has been covered. The evaluation in the box in line 11 of Algorithm 1 is hence defined as $!covered(p, \neg b)$, which means we will switch the predicate if the other branch has not been covered. Once we decide to switch an additional predicate, the fitness function is updated to reflect the new coverage (line 12). The number of executions needed is hence $O(n)$ with n the number instructions in the binary.
- *Quadratic Search.* In applications such as identifying indirect call targets, which is a very important challenge in binary analysis, simply covering all instructions may not be sufficient, we may need to cover paths that may lead to indirect calls or generate different indirect call targets. We hence define $\mathcal{F}$ as a set *icalls* to keep the set of the indirect call sites and potential indirect call targets that have been discovered by all the explored paths. The evaluation in line 11 is hence to test if cardinality of *icall* grows with the currently explored path. If so, the execution is considered important and all eligible unique predicates (not instances) in the execution are further explored. The complexity is $O(n^2)$ with n the number of instructions. X-Force can also limit the

quadratic search within a function.

- *Exponential Search.* If we simply set the evaluation in the line 12 to true, the algorithm performs exponential search because it will explore each possible combination. In practice, we cannot afford such search. However, X-Force provides the capability for the user to limit such exponential search within a sub-range of the binary.

Taint Analysis to Reduce Search Space. An observation is that we do not have to force-set predicates in low-level utility methods, because their branch outcomes are usually not affected by any input. Hence in X-Force, we use taint analysis to track if a predicate is related to program input. X-Force will only force branch outcomes of those tainted predicates. Since this is a standard technique, we omit its details.

4 Practical Challenges

In this section, we discuss how we address some prominent challenges in handling real world executables.

Jump Tables. In our previous discussion, we assume control transfer is only through simple predicates. In reality, jump tables allow a jump instruction to have more than two branches. Jump tables are widely used. They are usually generated from `switch` statements in the source code level. In X-Force, we leverage existing jump table reverse engineering techniques [21] to recover the jump table for each indirect jump. Our exploration algorithm then tries to explore all possible targets in the table.

Handling Loops and Recursions. Since X-Force may corrupt variables, if a loop bound or loop index is corrupted, an (incorrect) infinite loop may result. Similarly, if X-Force forces the predicate that guards the termination of some recursive function call, infinite recursion may result. To handle infinite loops, X-Force leverages taint analysis to determine if a loop bound or loop index is computed from input. If so, it resets the loop bound/index value to a pre-defined constant. To handle infinite recursion, X-Force constantly monitors the call stack. If the stack becomes too deep, X-Force further checks if there are cyclic call paths within the call stack. If cyclic paths are detected, X-Force skips calling into that function by simulating a "ret" instruction.

Protecting Stack Memory. Our early discussion on memory safety focused on protecting heap memory. However, it is equally important to protect stack memory. Particularly, the return address of a function invocation and the stack frame base address of the caller are stored on stack upon the invocation. They are restored when the callee returns. Since X-Force may corrupt variable values that affect stack accesses, such critical data could be undesirably over-written. We hence need to protect stack memory as well. However, we cannot simply prevent any stack write beyond the current frame. The strategy of X-Force is to prevent any stack writes that originate in the current stack-frame to go beyond the current frame. Specifically, when a stack write attempts to over-write the return address, the write is skipped. Furthermore, the instruction is flagged. Any later instances of the instruction that access a stack address beyond the current stack-frame are also skipped. The flags are cleared when the callee returns.

Handling Library Function Calls. The default strategy of X-Force is to avoid switching predicates inside library calls as our interest falls in user code. X-Force handles the following library functions in some special ways.

- *I/O functions.* X-Force skips all output calls and most input calls except file inputs. X-Force provides wrappers for file opens and file reads. If the file to open does not exist, X-Force skips calling the real file open and returns a special file handler. Upon file reads, if the file handler has the special value, it returns without reading the file such that the input buffer contains random values. Supporting file reads allows X-Force to avoid unnecessary failure recovery and path exploration if the demanded files are available.
- *Memory manipulation functions.* To support memory safety, X-Force wraps memory allocation and de-allocation. For memory copy functions such as `memcpy()` and `strcpy()`, the X-Force wrappers first determine the validity of the copy operation, e.g., the source and target address ranges must have been allocated, must not overlap with any critical stack addresses. If necessary, on-demand allocation is performed before calling the real function. This eliminates the need of memory safety monitoring, linear set tracing, and memory error recovery inside these functions, which could be quite heavy-weight due to the special structure of these functions. For example, `memcpy()` copies individual addresses one by one and these addresses are linearly correlated as they are computed through pointer manipulation, leading to very large linear sets.

For statically linked executables, X-Force relies on IDA-Pro to recognize library functions in a pre-processing step. IDA leverages a large signature dictionary to recognize library functions with very good accuracy. For functions that are not recognized by IDA, X-Force executes them as user code.

Handling Threads. Some programs spawn additional threads during their execution. It is difficult for X-Force to model multiple threads into a single execution since the order of their execution is nondeterministic. If we

simply skip the thread creation library functions such as *CreateThread()* and *beginthread()*, the functions in the thread could not be covered. To solve this problem, we adopt a simple yet effective approach of serializing the execution of threads. The calls to thread creation library functions are replaced with direct function calls to the starting functions of threads, which avoid creating multiple threads and guarantees code coverage at the same time. Note that as a result, X-Force is incapable of analyzing behavior that is sensitive to schedules. We will leave it to our future work.

5 Evaluation

X-Force is implemented in PIN. It supports WIN32 executables. In this section, we use three application case studies to demonstrate the power of X-Force.

Table 4: Detailed Coverage Comparison with Dynamic Analysis

	Input Union	X-Force	Input Union ∩ X-Force	Input Union \ X-Force	X-Force \ Input Union
164.gzip	3601	5075	3601	0	1474
175.vpr	19398	29218	19398	0	9820
176.gcc	157451	227546	157451	0	70095
181.mcf	1622	1935	1622	0	313
186.crafty	27811	42763	27811	0	14952
197.parser	17339	23135	17339	0	5796
252.eon	15580	27224	15580	0	11644
253.perlbmk	55964	33643	27003	28961	6640
254.gap	37564	110066	37564	0	72502
255.vortex	53798	101207	53798	0	47409
256.bzip2	3612	4830	3612	0	1218
300.twolf	19996	41935	19996	0	21939

5.1 Control Flow Graph (CFG) and Call Graph (CG) Construction

Construction of CFG and CG is a basic but highly challenging task for binary analysis, especially the identification of indirect call targets. In the first case study, we apply X-Force to construct CFGs and CGs for stripped SPECINT 2000 binaries. We also evaluate the performance of X-Force in this study. To construct CFGs and CGs, we use X-Force to explore execution paths and record all the instructions, control flow edges, and call edges, including indirect jump and indirect call edges. The exploration algorithm is a combination of linear search and quadratic search (Section 3.2). Quadratic search is limited to functions that contain indirect calls or encounter values that look like function pointers.

We compare X-Force results with four other approaches: (1) IDA-Pro; (2) Execute all the test cases provided in SPEC and union the CFGs and CGs observed for each program (i.e., dynamic analysis); (3) Static CG construction using LLVM on SPEC source code (i.e., static analysis) [3]. (4) Dynamic CFG construction using a symbolic execution system S2E [10]. We could not compare with CodeSurfer-X86 [2], which can also generate CFG/CG for executables based on static analysis, because it is not available through commercial or academic license.

Part of the results is presented in Table 3. Columns 2-4 present the instructions that are covered by the different approaches. Particularly, the second column shows the number of instructions recognized by IDA. The third column shows those that are executed by concrete input runs. Columns 5-8 show the indirect call edges recognized by the different approaches[4]. The last five columns show internal data of X-Force.

From the coverage data, we observe that X-Force could cover a lot more instructions than dynamic analysis except `253.perlbmk`. Note that the dynamic analysis results are acquired using all the test, training and reference inputs in SPEC, which are supposed to provide good coverage. Table 4 presents more detailed coverage comparison with dynamic analysis. Observe that X-Force covers all the instructions that are covered by natural runs for all benchmarks except `253.perlbmk`, which we will explain later. X-Force could cover most of the instructions identified by IDA except `252.eon` and `253.perlbmk`. We have manually inspected the differences between the IDA and X-Force coverage. For most programs except `253.perlbmk`, the differences are caused by part of the code in those binaries being unreachable. In other words, they are dead code that cannot be executed by any input. Since IDA simply scans the code body to construct CFG and CG, it reports all instructions it could find including the unreachable ones.

Table 5: Detailed Indirect Call Edges Identification Comparison with Dynamic Analysis

	Input Union	X-Force	Input Union ∩ X-Force	Input Union \ X-Force	X-Force \ Input Union
164.gzip	2	2	2	0	0
176.gcc	169	1720	169	0	1551
252.eon	60	121	60	0	61
253.perlbmk	225	151	103	122	48
254.gap	1103	20485	1103	0	19382
255.vortex	28	30	28	0	2

Indirect call edge identification is very challenging in binary analysis as a call site may have multiple call targets depending on execution states, which are usually difficult to cover or abstract. Some of them are dependent on states related to multiple procedures. Note that there does not exist an oracle that can provide the ground truth for the set of real indirect call edges. From the results, we could observe that LLVM's indirect call identification algorithm generates a large number of edges, much more than X-Force. However, we confirm that most of them are bogus because the LLVM algorithm simply relies on method signatures to identify possible targets and

[3] We cannot compare LLVM CFGs with X-Force CFGs as LLVM CFGs are not represented at the instruction level.

[4] Direct jump and call edges are easy to identify and elided.

Table 3: CFG and CG Construction Results.

	Coverage			Indirect Call Edge				X-Force Internals				
	IDA-Pro	Input Union	X-Force	IDA-Pro	Input Union	LLVM	X-Force	Time (s)	# of Runs	Avg. # of Exp.	Avg./Max. Linear Set Size	Switched/Total # of predicates
164.gzip	7913	3601	5075	0	2	2	2	704	246	10	2.9/36	2.1/1291
175.vpr	31847	19409	29218	0	0	0	0	8725	1849	49	2.8/19	4.7/2164
176.gcc	310277	157451	227546	25	169	9141	1720	173241	26606	95	4.5/265	12.9/29847
181.mcf	2184	1622	1935	0	0	0	0	129	113	10	3.1/23	4.3/153
186.crafty	43327	27811	42763	0	0	0	0	43995	2496	0.4	2.6/9	8.0/62582
197.parser	25532	17339	23135	0	0	0	0	3424	1820	8	2.5/17	6.4/944
252.eon	70592	15580	27224	0	60	28802	121	6379	2091	4	2.3/10	4.1/3146
253.perlbmk	132264	55964	33643	24	225	-	151	7137	843	0.8	3.5/40	8.3/9535
254.gap	113410	37564	110066	2	1103	187155	20470	50745	7319	1353	30.0/1846	6.0/173316
255.vortex	132053	53798	101207	0	28	340	30	34776	8566	13	2.9/33	7.3/2548
256.bzip2	5761	3612	4830	0	0	0	0	557	209	5	3.3/15	1.4/7001
300.twolf	46556	19996	41935	0	0	0	0	10043	2825	17	2.6/8	5.4/1322

hence is too conservative. X-Force could recognize a lot more indirect call edges than dynamic analysis. The detailed comparison in Table 5 shows that the X-Force results cover all the dynamic results and have many more edges, except *253.perlbmk*. We have manually inspected a random set of the selected edges that are reported by X-Force but not the dynamic analysis and confirmed that they are feasible. From the results in Table 3, IDA can hardly resolve any indirect call edges.

Table 6: Result of using S2E to analyze SPEC programs

	Basic Block Coverage	Function Block Coverage	Touched Functions	Fully Covered Functions	Number of Paths
164.gzip	768/2240(34%)	768/1294(59%)	62/186(33%)	21/186(11%)	134
176.gcc	740/46487(1%)	740/1468(50%)	62/1398(4%)	19/1398(1%)	261
252.eon	64/2830(2%)	64/101(63%)	19/649(2%)	13/649(2%)	33
253.perlbmk	1708/37384(4%)	1708/6912(24%)	134/1510(8%)	27/1510(1%)	329
254.gap	1235/28871(4%)	1235/3136(39%)	80/941(8%)	21/941(2%)	29
255.vortex	10933/35979(30%)	10933/20822(52%)	437/1031(42%)	21/1031(2%)	9

We also use S2E to analyze the six SPECINT 2000 programs that contain indirect calls. The four programs other than `252.eon` and `255.vortex` read input from *stdin*, so we use the *s2ecmd* utility tool provided by S2E to write 64 bytes to *stdout* and pipe the symbolic bytes into these programs. We run each program in S2E and use the `ExecutionTracer` plugin to record the execution trace. We use the IDA scripts provided by S2E to extract information of basic blocks and functions from the binaries, and then use the `coverage` tool provided by S2E to calculate the result.

The result is shown in Table 6. The columns show the following metrics from left to right: (1) coverage of basic blocks; (2) coverage of basic blocks when excluding the basic blocks in those functions that are not executed; (3) coverage of functions; (4) percentage of fully-covered functions; (5) the number of different paths that S2E explored. Observe that the coverage is much lower than X-Force in general. `176.gcc`, `253.perlbmk` and `254.gap` are parsers/compilers. They have poor coverage on S2E because they get stuck in the parsing loops/automatas, whose termination conditions are dependent on the symbolic input. Regarding `255.vortex`, S2E fails to solve the constraints when an indirect jump uses the symbolic variable as the index of jump table. As a result, S2E fails to identify most of the indirect call edges due to the failure of creating different objects. In `252.eon`, S2E fails to solve the constraints of the input file format, which must contain a specific string as header. The program throws exception and terminates quickly, which leads to poor coverage.

`253.perlbmk` is a difficult case for X-Force. It parses perl source code to generate syntax trees. The indirect call targets are stored in the nodes of syntax trees. However, since the syntax tree construction is driven by finite automata, path coverage does not seem to be able to cover enough states in the automata to generate enough syntax trees of various forms. A few other benchmarks such as `176.gcc` and `254.gap` also leverage automata based parsers, however their indirect call targets are not so closely-coupled with the state of the automata and hence X-Force can still get good coverage. We will leave it to our future work to address this problem.

The last five columns show some statistics of X-Force. The run time and the number of explorations are largely linear regarding the number of instructions except for a small number of functions on which quadratic search is performed. Some take a long time (e.g., close to 50 hours for `176.gcc`) due to their complexity. The average number of exceptions is the number of exceptions encountered and recovered from in each execution (e.g. memory exceptions, division by zero). The numbers are smaller than we expected given that we execute these programs without any inputs and switch branch outcomes. It shows that our exception recovery could effectively prevent cascading exceptions. The linear set sizes are manageable. The last column shows the average number of switched predicates versus the average number of predicate instances in total in an execution. It shows that X-Force may violate path feasibility only in a very small part of execution. The performance overhead of X-Force compared to the vanilla PIN is 473 times on average. It is measured by comparing the number of instructions that could be executed by X-Force and the vanilla PIN within the same amount of time.

5.2 Malware Analysis

One common approach to understanding the behavior of an unknown malware sample is by looking at the library calls it makes. This could be done by static, dynamic or symbolic analysis; however, they all have limitations.

Table 7: Result of using X-Force for malware analysis compared with IDA Pro and native run.

Name	MD5	File Size(KB)	Number of Library Functions			Number of Library Call Sites			No. of Runs in X-Force
			IDA Pro	Native Run	X-Force	IDA Pro	Native Run	X-Force	
dg003.exe	4ec0027bef4d7e1786a04d021fa8a67f	192	147	129	252	808	546	1750	800
Win32/PWSteal.F	04eb2e58a145462334f849791bc75d18	20	7	21	42	9	28	94	30
APT1.DAIRY	995442f722cc037885335340fc297ea0	19	90	40	100	213	68	236	121
APT1.GREENCAT	0c5e9f564115bfcbee66377a829de55f	14.5	66	26	64	303	114	302	112
APT1.HELAUTO	47e7f92419eb4b98ff4124c3ca11b738	8.5	41	16	39	109	33	109	30
APT1.STARSYPOUND	1f2eb7b090018d975e6d9b40868c94ca	7	37	14	36	80	15	80	25
APT1.WARP	36cd49ad631e99125a3bb2786e405cea	45.5	77	47	79	495	156	414	221
APT1.NEWSREEL	2c49f47c98203b110799ab622265f4ef	21	67	31	67	189	49	192	93
APT1.GOGGLES	57f98d16ac439a11012860f88db21831	10.5	35	21	36	127	45	131	42
APT1.BOUNCER	6ebd05a02459d3b22a9d4a79b8626bf1	56	11	16	97	24	39	562	298

Static analysis could not obtain the parameters of library calls that are dynamically computed and is infeasible when the sample is packed or obfuscated. Traditional dynamic analysis can obtain parameters and is immune to packing and obfuscation, however, it could only explore some of the execution paths depending on the input and the environment. Unfortunately, the input is usually unknown for malware. Symbolic analysis, while being able to construct input according to path conditions, has difficulty in handling complex or packed binaries.

X-Force overcomes these problems as traditional dynamic analysis could be built upon X-Force to explore various execution paths without providing any inputs or the environment. In this case study, we demonstrate the use of a library call analysis system we built on top of X-Force to analyze real-world malware samples.

When we implement library call analysis on top of X-Force, we slightly adjust X-Force to make it suitable for handling malware: (1) We enable the concrete execution of most library functions including output functions because many packers use output functions (e.g. `RtlDecompressBuffer()`) to unpack code. We continue to skip some library calls such as `Sleep()` and `DeleteFile()`; (2) We intercept a few functions that allocate memory and change page attributes, such as `VirtualAlloc()` and `VirtualProtect()`. This is for tracking the memory areas of code and data which keep changing at runtime due to self-modifying and dynamically generated code.

Given a malware sample, we use X-Force to explore its paths. We use the linear search algorithm (Section 3.2) as it provides a good balance between efficiency and coverage. During each execution, we record a trace of function calls. For library calls, we also record the parameter values. The trace is then transformed into an interprocedural flow graph that has control transfer instructions, including jumps and calls, as its nodes, and control-flow/call edges as its edges. The parameters of library calls are also annotated on the graph. The graphs generated in multiple executions are unioned to produce the final graph. We then manually inspect the final graphs to understand malware behavior.

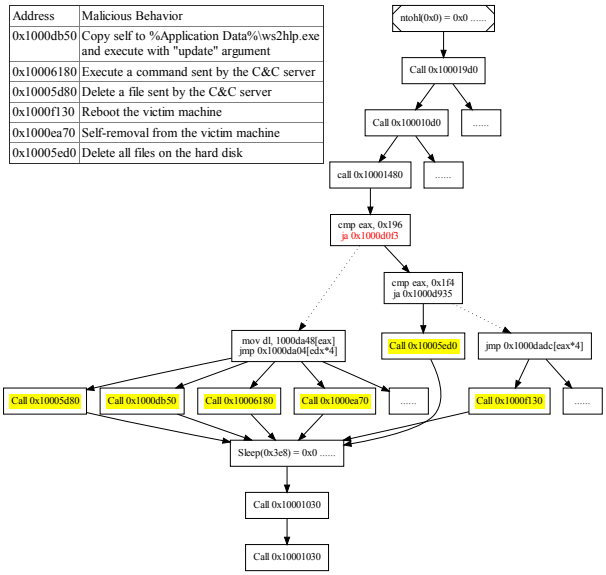

Figure 5: The flow graph of the function at *0x1000c630* generated by X-Force when analyzing dg003.exe.

We evaluate our system on 10 real-world malware samples which are either wild-captured virus/trojan or APT samples described in [9]. Since our analysis focuses on library calls, we choose the number of identified library functions and the total number of their call sites as the evaluation metric[5]. We also compare our results with IDA-Pro and the native run. In IDA, library functions are identified from the import table; the call sites are identified by scanning the disassemblies. In the native run, we execute the malware without any arguments and record the library calls using a PIN tool.

The results are shown in Table 7. We can see that for packed or obfuscated samples such as *dg003.exe*, *Win32/PWSteal.F*, *APT1.DAIRY*, and *APT1.BOUNCER*, IDA gets fewer library functions and call sites compared to X-Force. For other samples that are not packed or obfuscated, since the executables could be properly disassembled, the metrics obtained in IDA and X-Force are

[5]We exclude the C/C++ runtime initialization functions which are only called before the main function.

very close. However, even in such cases, static analysis is insufficient to understand the malicious behavior because it does not show the values of the library function parameters. Compared to the native run method, X-Force can identify more library functions and call sites.

Next, we present detailed analysis for two representative samples.

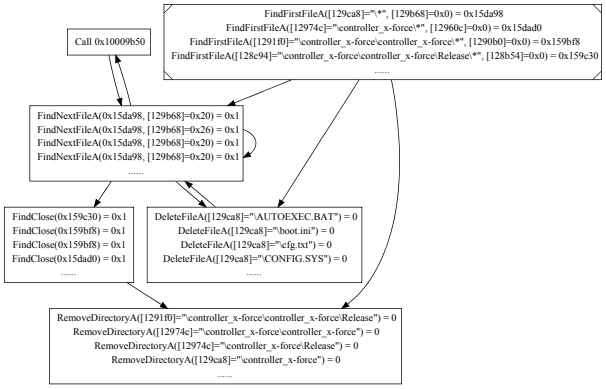

Figure 6: The flow graph of the function at *0x10009b50* in dg003.exe that delete all files on the hard disk.

Dg003.exe. This is a typical APT malware sample that features multi-staged, condition-guarded and environment-specific payload. In the first stage, the malware extracts a DLL which it carries as its resource, packs the DLL in memory using a proprietary algorithm and writes the packed DLL to the disk. In the second stage, the packed DLL is loaded, unpacks itself in memory and executes the main payload.

There is a previous report [26] in which the analysts used both static and dynamic analyses to analyze this sample. To perform static analysis using IDA Pro, they manually extract and unpack the DLL. This requires reverse engineering the unpacking algorithm, which could be both time consuming and difficult. Our system avoids such trouble by concretely executing the unpacking routine which performs the unpacking for us. Compared with their dynamic analysis, it takes X-Force about 5 hours to finish 800 executions to explore all paths in both the first and second stages of the malware. After that, the traces are transformed into a flow graph containing 378 functions. Our system is able to discover a set of malicious behaviors that are *NOT* mentioned in the previous report. As shown in Fig. 5, each highlighted function call in the graph corresponds to a previously unrevealed malicious behavior. Each behavior is identified using the library calls made in the corresponding function. For example, as shown in Fig. 6, the library calls and the parameters in the function at *0x10009b50* show that it recursively enumerates and deletes files and directories starting from the root directory, which indicates its behavior is to delete all files on the disk.

In Fig. 5 we can see that the common dominator of all these function calls (highlighted in red color) determines if the value of *eax* register is larger than 0x196. With taint analysis in X-Force, we find that the value of the *eax* register is related to an input which is a buffer in a previous *recv* library function call. This indicates it represents the command ID sent by the C&C server, which leads to the execution of different malicious behaviors. Hence, we suspect that the previous analysts missed some behaviors because the C&C server only sent part of the possible commands at the time they ran the malware. We also find that the buffer in the *recv* function call is translated to the command ID using a private decryption algorithm, so it would be infeasible for symbolic analysis to solve the constraints and construct a valid input. We also want to point out that at the time we perform the analysis, the C&C server of this malware is already inactive; we would not be able to discover these malicious behaviors, had we not used X-Force.

Win32/PWSteal.F. Before trying X-Force on this sample, we first try static analysis using IDA-Pro. Surprisingly, this sample does not import any suspicious library function; not even a function that could perform I/O (e.g. read/write file, registry or network socket). The `LoadLibrary()` and `GetProcAddress()` functions are not imported either, which means the common approach of dynamically loading libraries is not used. The strings in the executable do not contain any DLL name or library function name either. This indicates the sample is equipped with advanced API obfuscation technique to thwart static analysis.

Since static analysis is infeasible, we submit the sample to the Anubis malware analysis platform for dynamic analysis. The result shows the malware does read some registry entries and files, however, none of them seems malicious. Hence, we feed the sample to our system in hopes of revealing its real intent. X-Force achieves full coverage after exploring 30 paths and generates a graph with 15 functions. By traversing the graph, we find that this malware aims at stealing the password that is stored by IE and Firefox in the victim's machine. It enumerates the registry entry that stores the encrypted auto-complete password for IE and calls library functions such as `CryptUnprotectData()` to decrypt the stored password. This is very similar to the attack mentioned in [1]. Regarding Firefox, it first gets the user name from `profiles.ini` under the Firefox application data directory, and then steals the password that is stored in the `signons*.txt` under the directory of the user name. The password is then uploaded to a remote FTP server using the file name `[Computer Name].[IP Address].txt`. Clearly, this sample finds the entry addresses of these library functions at runtime using some obfuscation techniques. X-Force allows us to identify the

malicious behavior without spending unnecessary time on reverse-engineering the API obfuscation.

Moreover, the flow graph also reveals the reason why Anubis missed the malicious behavior: the malware performs environment checks to make sure the targets exist before trying to attack. For example, in the function where the malware steals password from IE, it will try to open the registry entry that contains the auto-complete password; if such entry does not exist or is empty, the malware will cease its operation and return from that function. Also, before it tries to steal password stored by Firefox, it will first try querying the installation directory of Firefox from registry to make sure the target program exists in the system. Such "prerequisites" are unlikely to be fulfilled in automated analysis systems as they are unpredictable. However, by force-executing through different paths, X-Force is able to get through these checks to reveal the real intent of the malware.

```
TYPE_1 func1(TYPE_2 arg1, TYPE_3 arg2) {
        TYPE_4 var1;
1       var1 = strlen (arg1);
2       if (arg2 >= var1)
3               return 0;
4       return arg1[arg2];
}
```

Figure 7: REWARDS example.

5.3 Type Reverse Engineering

Researchers have proposed techniques to reverse engineer variable and data structure types for stripped binaries [30, 39, 25]. The reverse engineered types can be used in forensic analysis and vulnerability detection. There are two common approaches. REWARDS [30] and HOWARD [39] leverage dynamic analysis. They can produce highly precise results but incompleteness is a prominent limitation – they cannot reverse engineer types of variables if such variables are not covered by executions. TIE [25] leverages static analysis and abstract interpretation such that it provides good coverage. However, it is challenging to apply the technique to large and complex binaries due to the cost of analysis.

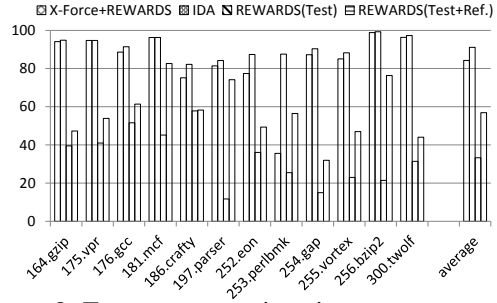

Figure 8: Type reverse engineering coverage results.

One advantage of X-Force is that the forced executions are essentially concrete executions such that existing dynamic analyses could be easily ported to X-Force to benefit from the good coverage. Therefore in the third case study, we port the implementation of REWARDS to X-Force. Given a binary executable and a few test inputs, REWARDS executes it while monitoring dataflow during execution. When execution reaches system or library calls, the types of the parameters of these calls are known. Such execution points are called *type sinks*. Through the dynamic dataflow during execution, such types could be propagated to variables that (transitively) contributed to the parameters in the past and to variables that are (transitively) dependent on these parameters.

Consider the example in Fig. 7. Assume `func1` is executed. After line 1, the type of `arg1` and `var1` get resolved using the interface of `strlen()`. So `TYPE_2` is `char *`, and `TYPE_4` is `unsigned int`. In line 2, `arg2` is compared with `var1`, implying they have the same type. Thus `TYPE_3` gets resolved as `unsigned int`. Later when line 4 gets executed, it returns `TYPE_1` which is resolved as `char` since `arg1` is of `char *`.

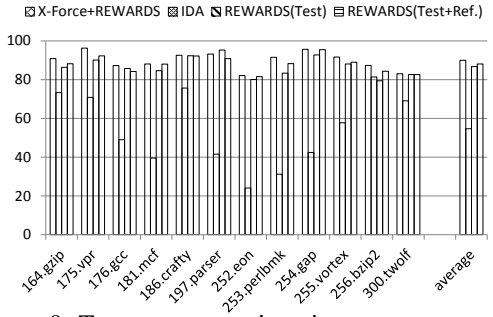

Figure 9: Type reverse engineering accuracy results.

Porting REWARDS to X-Force requires very little modification of either the REWARDS or the X-Force systems as they only interface through the (forced) concrete executions. Facilitated by X-Force, REWARDS is able to run legacy binaries and COTS binaries without any inputs. In our experiment, we run the new system on the 12 SPEC2000 INT binaries. They are a lot more complex than the Linux core-util programs used in the original paper [30]. To acquire the ground truth, we compile the programs with the option of generating debugging symbols as PDB files, and use DIA SDK to read the type information from the PDB files.

We evaluate the system in terms of both coverage and accuracy. Coverage means the percentage of variables in the program that have been executed by our system. Accuracy is the percentage of the covered variables whose types are correctly reverse engineered. From Fig. 8, the average coverage is around 84%. The coverage heavily relies on the code coverage of X-Force. Recall that these programs have non-trivial portion of unreachable code. The variables in those code regions cannot be reverse engineered by our system. From Fig.9, the average accuracy is about 90%. The majority of type inference

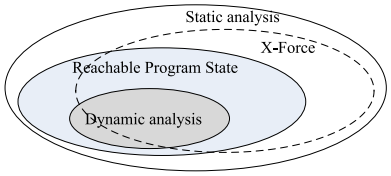

Figure 10: Essence of X-Force.

failures is caused by the fact that the variables are not related to any *type sink*.

We also compare with IDA and the original REWARDS. IDA has a static type inference algorithm that works in a similar fashion. When we run the original REWARDS, we have two configurations: (1) use the test input only (1 input per program) and (2) use both the test and the reference inputs (around 4 inputs per program). From Fig. 8 and Fig. 9, our system has much better accuracy than IDA (90% vs. 55% on average) and better coverage than the original REWARDS, i.e., 84% vs. 57% (test+reference) or 34% (test input only). The better accuracy than IDA is achieved by the more precise modeling of behavior difficult for static analysis, such as heap accesses and indirect calls and jumps.

6 Discussion and Future Work

X-Force is intended to be a *practical* solution for analyzing unknown (malicious) binaries without requiring any source code or inputs. Hence, X-Force trades soundness and completeness for practicality. It is unsound as it could explore infeasible paths. It is incomplete as it cannot afford exploring all paths. Figure 10 shows how X-Force compares with static and dynamic analysis: The "Reachable Program State" oval denotes all states that can be reached through possible program inputs – the ideal coverage for program analysis. Static analyses often make conservative approximations such that they yield *over-approximate* coverage. Dynamic analyses analyze a number of real executions and hence yield *under-approximate* results. X-Force explores a larger set of executions than dynamic analyses. Since X-Force makes unsound approximations, its results may be invalid (i.e., outside the ideal oval). Furthermore, it is incomplete as its results may not cover the ideal ones.

However, we argue that X-Force is still of importance in practice: (1) There are many security applications whose analysis results are not so sensitive to paths, such as the three studies in this paper. As such, path infeasibility may not affect the results much. However, having concrete states in path exploration is still critical in these applications such that an execution based approach like X-Force is suitable; (2) Only a very small percentage of predicates are switched (Section 5.1) in X-Force. Execution is allowed to proceed naturally in most predicates, respecting path feasibility. According to our observations, most of the predicates that got switched in linear search are those checking if the program has been provided the needed parameters, if files are properly opened, and if certain environmental configurations are correctly set-up; (3) In X-Force, taint analysis is used to identify predicates that are affected by inputs and only such predicates are eligible for switching.

Moreover, X-Force allows users to (1) *rapidly* explore the behaviors of any (unknown) binary as it simply executes the binary (without solving constraints); (2) handle binaries in a much *broader* spectrum (e.g., large, packed, or obfuscated binaries); (3) easily port or develop dynamic analysis on X-Force as the executions in X-Force are no different from regular concrete executions.

Future Work. We believe this paper is just an initial step in developing a unique type of program analysis different from the traditional static, dynamic, and symbolic analysis. We have a number of tasks in our future research agenda.

- While X-Force simply forces the branch outcomes of a few predicates without considering their feasibility, we suspect that there is a chance in practice the forced paths are indeed feasible in many cases. Note that the likelihood of infeasibility is not high if the forced predicates are not closely correlated. We plan to use a symbolic analysis engine that models the path conditions along the forced paths to observe how often they are infeasible.

- We develop 3 exploration algorithms in this paper. From the evaluation data on the SPECINT2000 programs, there are cases (e.g., perlbmk) that the current exploration algorithms cannot handle well. More effective algorithms, for example, based on modeling functions behaviors and caching previous exploration choices, will be developed.

- We currently handle multi-threaded programs by serializing their executions. In the future, we will explore forcing real concurrent executions. We envision this has to be integrated with flipping schedule decisions, which is a standard technique in exploring concurrent execution state. How to handle the enlarged state space and the potentially introduced infeasible thread schedules will be the new challenges.

- The current system is implemented as a tool on top of PIN. To build a tool that makes use of X-Force, for example REWARDS, the implementation of the additional tool is currently mixed with X-Force. They are compiled together to a single PIN-tool. We aim to make X-Force transparent to dynamic analysis developers by providing an PIN-like interface. Ideally, existing PIN-tools can be easily

ported to X-Force to benefit from the large number of executions provided by the X-Force engine.

- We also plan to port the core X-Force engine to other platforms such as mobile and HTML5 platforms.

7 Related Work

Researchers proposed to force branch outcomes for patching software failures in [51]. Hardware support was proposed to facilitate path forcing in [31]. Both require source code and concrete program inputs. Branch outcomes are forced to explore paths of binary programs in [48] to construct control flow graphs. The technique does not model any heap behavior. Moreover, it skips all library calls. Similar techniques are proposed to expose hidden behavior in Android apps [22, 45]. These techniques randomly determine each branch's outcome, posing the challenge of excessive infeasible paths. Forced execution was also proposed to identify kernel-level rootkits [46]. It completely disregards branch outcomes during execution and performs simple depth-first search. None of these techniques performs exception recovery and instead simply terminates executions when exceptions arise. Constraint solving was used in exploring execution paths to expose malware behavior in [33, 6]. They require concrete inputs to begin with and then mutate such inputs to explore different paths.

X-Force is related to static binary analysis [21, 3, 25, 42, 41], dynamic binary analysis [30, 39, 24] and symbolic binary analysis [10, 40]. We have discussed their differences from X-Force in Section 6, which are also supported by our empirical results in Section 5. X-Force is also related to failure oblivious computing [36] and on-the-fly exception recovery [34], which are used for failure tolerance and debugging and require source code.

8 Conclusion

We develop a novel binary analysis engine X-Force, which forces a binary to execute without any inputs or the needed environment. It systematically forces the branch outcomes at a small number of predicates to explore different paths. It can recover from exceptions by allocating memory on-demand and fixing correlated pointers accordingly. Our experiments on three security applications show that X-Force has similar precision as dynamic analysis but much better coverage due to the capability of exploring many paths with any inputs.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. This work was motivated in part by the earlier research of Dr. Vinod Yegneswaran on brute-force malware execution and analysis. His influence and support is gratefully acknowledged. This research has been supported, in part, by DARPA under Contract 12011593 and by a gift from Cisco Systems. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors.

References

[1] Exposing the password secrets of internet explorer. `http://securityxploded.com/iepasswordsecrets.php`.

[2] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum. Codesurfer/x86–a platform for analyzing x86 executables. In *Proceedings of International Conference on Compiler Construction (CC)*, 2005.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of Intenational Conference on Compiler Construction (CC)*, 2004.

[4] D. Balzarotti, M. Cova, C. Karlberger, E. Kirda, C. Kruegel, and G. Vigna. Efficient detection of split personalities in malware. In *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2010.

[5] R. R. Branco, G. N. Barbosa, and P. D. Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. Blackhat USA'12.

[6] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. 2008.

[7] J. Caballero and D. Song. Polyglot: Automatic extraction of protocol format using dynamic binary analysis. In *Proceedings of the 14th ACM Conference on Computer and and Communications Security (CCS)*, 2007.

[8] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI)*, 2008.

[9] M. I. Center. Apt1: Exposing one of chinas cyber espionage units. Technical report, 2013.

[10] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2011.

[11] C. Csallner and Y. Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, pages 245–254, 2006.

[12] W. Cui, J. Kannan, and H. J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *Proceedings of the 16th USENIX Security Symposium (Security)*, 2007.

[13] W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[14] Z. Deng, X. Zhang, and D. Xu. Bistro: Binary component extraction and embedding for software security applications. In *18th European Symposium on Research in Computer Security (ESORICS)*, 2013.

[15] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*, 2008.

[16] N. Falliere, L. Murchu, and E. Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 2011.

[17] P. Ferrie. Attacks on virtual machine emulators. *Symantec Advanced Threat Research*, 2006.

[18] P. Ferrie. Attacks on more virtual machine emulators. *Symantec Technology Exchange*, 2007.

[19] FireEye. Advanced targeted attacks: How to protect against the new generation of cyber attacks. In *White Paper*, 2013.

[20] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[21] Hex-Rays. Ida pro disassembler. `http://www.hex-rays.com/products/ida/index.shtml`.

[22] R. Johnson and A. Stavrou. Forced-path execution for android applications on x86 platforms. Technical report, Technical Report, Computer Science Department, George Mason University, 2013.

[23] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th USENIX Security Symposium (Security)*, 2009.

[24] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *2010 IEEE Symposium on Security and Privacy (SP)*, pages 29–44, 2010.

[25] J. Lee, T. Avgerinos, and D. Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[26] F. Li. A detailed analysis of an advanced persistent threat malware. *SANS Institute*, 2011.

[27] J. Lim, T. Reps, and B. Liblit. Extracting file formats from executables. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE)*, 2006.

[28] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[29] Z. Lin and X. Zhang. Deriving input syntactic structure from execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2008.

[30] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[31] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO)*, 2006.

[32] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected means of protocol inference. In *Proceedings of the 6th ACM SIGCOMM*

on Internet measurement (IMC), pages 313–326, 2006.

[33] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy (SP)*, pages 231–245, 2007.

[34] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3), 2007.

[35] T. Raffetseder, C. Krügel, and E. Kirda. Detecting system emulators. In *Proceedings of the 10th international conference on Information Security (ISC)*. 2007.

[36] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee, Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, 2004.

[37] N. Riva and F. Falcón. Dynamic binary instrumentation frameworks: I know you're there spying on me. In *RECON Conference*, 2012.

[38] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-13)*, 2005.

[39] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

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

[41] B. D. Sutter, B. D. Bus, K. D. Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *Proceedings of Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2000.

[42] H. Theiling. Extracting safe and precise control flow from binaries. In *Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA)*, 2000.

[43] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *2006 IEEE Symposium on Security and Privacy (SP)*, 2006.

[44] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. Reformat: Automatic reverse engineering of encrypted messages. In *Proceedings of 14th European Symposium on Research in Computer Security (ESORICS)*, 2009.

[45] Z. Wang, R. Johnson, R. Murmuria, and A. Stavrou. Exposing security risks for commercial mobile devices. In *Proceedings of the 6th international conference on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security (MMM-ACNS)*, pages 3–21, 2012.

[46] J. Wilhelm and T.-c. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proceedings of the 10th international conference on Recent advances in intrusion detection (RAID)*, pages 219–235, 2007.

[47] G. Wondracek, P. Milani, C. Kruegel, and E. Kirda. Automatic network protocol analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[48] L. Xu, F. Sun, and Z. Su. Constructing precise control flow graphs from binaries. Technical report, Technical Report CSE-2009-27, Department of Computer Science, UC Davis, 2009.

[49] L.-K. Yan, M. Jayachandra, M. Zhang, and H. Yin. V2e: Combing hardware virtualization and software emulation for transparent and extensible malware analysis. In *8th Annual International Conference on Virtual Execution Environments (VEE)*, 2012.

[50] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, 2007.

[51] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of ACM/IEEE International Conference on Software Engineering (ICSE)*, 2006.