CodeXt: Automatic Extraction of Obfuscated Attack Code from Memory Dump

Ryan Farley and Xinyuan Wang
Department of Computer Science
George Mason University
Fairfax, VA 22030, USA
email: {rfarley3, xwangc}@gmu.edu

Abstract. In this paper, we present *CodeXt*—a novel malware code extraction framework built upon selective symbolic execution (S2E). Upon real-time detection of the attack, CodeXt is able to automatically and accurately pinpoint the exact start and boundaries of the attack code even if it is mingled with random bytes in the memory dump. CodeXt has a generic way of handling self-modifying code and multiple layers of encoding, and it can automatically extract the complete hidden and transient code protected by multiple layers of sophisticated encoders without using any signature or pattern of the decoder. To the best of our knowledge, CodeXt is the first tool that can automatically extract code protected by Metasploit's polymorphic xor additive feedback encoder Shikata-Ga-Nai, as well as transient code protected by multi-layer incremental encoding.

Keywords: Malware Forensics, Binary Analysis, Symbolic Execution.

1 Introduction

Automatically recovering malware attack code is critical to improving effective malware analysis, forensics, and reverse engineering. Existing methods involve substantial manual effort. Given the sheer number of new malwares that arrive every year, it would be invaluable to be able to automate recovery (from the run-time memory) upon detection of an attack.

First, we must automatically pinpoint the exact start and boundaries of the attack code, possibly spread across disjoint segments and intermingled in random surrounding data and/or code bytes within memory, per Figure 1. Second, the attack code can be easily obfuscated with self-modification such as encoding and packing which renders static analysis ineffective. Third, the attack code may never reveal its complete unpacked version in run-time memory at any singular moment, such as multiple layers of decoding or unpacking where each layer only extracts a portion of the final attack code, such as in Figure 2. Such an incremental decoding makes it very difficult to automatically recover the complete attack code even if one can dump run-time memory at any time.

A number of approaches [26, 9, 31] detect attack code in network traffic. These methods can detect some snippets of code from the packet data, but neither determine the exact start, boundaries, nor recover the complete attack code. Existing dynamic analysis based unpacking approaches (e.g., PolyUnpack [21], Renovo [14], OmniUnpack [19]) recover hidden code from packed executables, and thus assume knowledge of the exact starting point and are not effective when

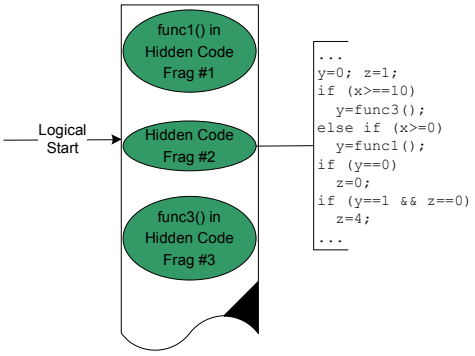

Fig. 1. Multiple Disjoint and Misaligned Code Fragments Mingled with Random Bytes

the packed byte code is mingled with random data or code. Traditional dynamic analysis approaches normally cover only one execution path and may miss hidden code and data in other (unexplored) paths. To the best best of our knowledge, no existing unpacking method has been shown to be able to recover the complete hidden code protected by the incremental encoding we present is this paper. Since existing software emulators (e.g., QEMU, S2E [10]) do not have full support of FPU instructions, they can neither execute nor recover the code protected by encoders (e.g., Shikata-Ga-Nai [2]) that use FPU instructions. This is the reason why SHELLOS [24] abandoned emulation for hardware virtualization.

In this paper, we seek to advance the current capability of automatic attack code extraction from run-time memory. We present *CodeXt*—a novel malware forensics framework based on selective symbolic execution (S2E) [10]. CodeXt uses two key techniques to achieve unprecedented capability in automatic attack code recovery: 1) combination of concrete and symbolic execution to recover potentially disjoint, misaligned, self-modifying code from all execution paths within a given memory range; 2) intelligent memory update clustering and multi-layer snapshots to recover all the code fragments of incremental decoding.

We have empirically validated the effectiveness of CodeXt with real world attack code and 9 well-known third party encoders (e.g., Shikata-Ga-Nai [2]) as well as 3 encoders (e.g., multi-layer incremental encoding) developed by ourselves. CodeXt is able to accurately locate the attack code that is mingled with random bytes and extract the complete (including transient) code hidden by all of the 12 encoders we have tested. To the best of our knowledge, CodeXt is the first tool that can automatically extract the code protected by Metasploit's polymorphic xor additive feedback encoder Shikata-Ga-Nai and the transient code protected by multi-layer incremental encoding.

2 Overview

Our approach does not seek to determine if a given piece of code is malicious or not, but rather extract hidden attack code from run-time memory upon real-time detection of malware or attack. CodeX thas the following advantageous features. 1) It can automatically identify the exact start and boundaries of all

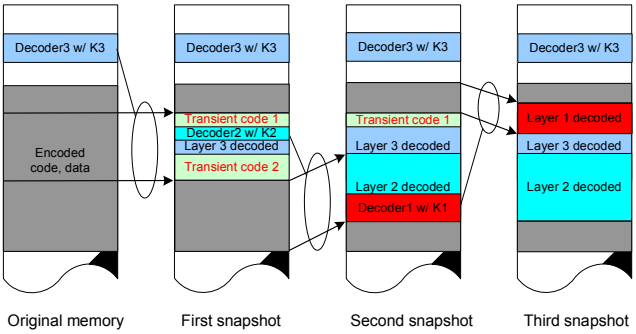

Fig. 2. Transient Code with Multiple Layer of Self-Modifying Code

hidden code fragments, even if they are mingled with random data in the run-time memory dump. 2) It can automatically recover the complete attack code, including transient code, protected by sophisticated self-modifying code such as multi-layer incremental encoding and/or packing with overlapped ranges and different keys. 3) It can automatically collect relevant intermediate results during multi-layered decoding, revealing obfuscations used at each layer. 4) It can merge all hidden code fragments into logically related collections. 5) It can recover the complete attack code protected by advanced polymorphic encoders that typically evade emulation, such as those that use FPU instructions or self-modify the current basic block of the run-time decoder. 6) It can validate the extracted hidden code via symbolic execution to verify that execution of extracted hidden code will lead to any detection conditions reported by the intrusion or malware detection system. 7) It is generic and does not rely on any signature or pattern of any particular decoder.

We assume there is some intrusion or malware detection system that can detect the execution of attack code in real-time (e.g., [18, 30]) and it will dump the memory around the instruction (e.g., system call) where the attack has been detected and other attack context information. Since many intrusion detection systems (e.g., [13, 32, 22, 28, 12, 18, 30]) use system calls to detect an attack, we assume the attack context information includes some system call triggered by the attack code and corresponding register values. We further assume the dumped memory is large enough to contain all hidden attack code present in the run-time memory when the attack was detected. To avoid the undecidability problem in unpacking determination [21], we assume there is no infinite loop in the the attack code and our system will terminate after a configurable maximum number of instructions have been executed.

2.1 Overall CodeXt Architecture

CodeXt uses a combination of symbolic and concrete execution during analysis. Symbolic execution allows CodeXt to pinpoint the exact code start and boundaries by exploring all the legitimate execution start points and paths. On the other hand, concrete execution enables CodeXt to handle potential dynamic binary transformation and self-modifying code. We choose to build CodeXt upon

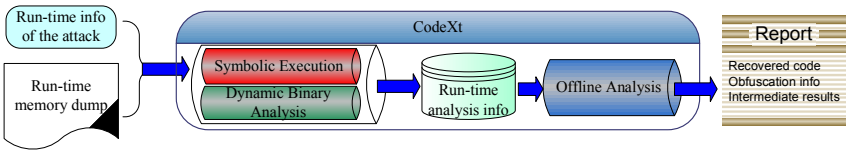

Fig. 3. Overall CodeXt Architecture

Selective Symbolic Execution (S2E) [10] which supports in-vivo multi-path analysis and allows us to execute any basic block either concretely with QEMU or symbolically with KLEE [8].

Figure 3 shows the overall architecture of CodeXt with an online and offline component. The online component consists of a number of S2E plugins which can monitor, track, and direct the selective symbolic execution of any given byte stream by exploring all execution paths from all offsets. It filters out impossible code snippets (e.g., invalid instruction, invalid memory access) and records those that are feasible and satisfy the attack context information given. To handle self-modifying code, CodeXt detects and records all instructions dynamically generated before execution and takes snapshots of each layer of self-modification outputting intermediate results. The offline component further analyzes the online results to derive the hidden code's start and boundaries.

3 Design and Implementation

How to Locate Hidden Code? We need to determine the existence of, exact start, and the boundaries of any hidden code from a given memory dump. The hidden code is usually mingled with random data/code. These constraints are different from that of traditional unpacking tools (e.g., PolyUnpack [21], Omni-Unpack [19] and Renovo [14]) which assume the execution start point is already known. We need to treat every offset in the memory dump as a possible logical start, or entry point, of the hidden code we are looking for. To leverage the system call information from the IDS, we have developed a S2E plugin to catch all the system calls triggered from within a given memory dump.

To reliably locate the logical start of the attack code in the memory dump, we use S2E and make the offset of the memory dump symbolic. This also employs S2E's efficient built-in state forking. To avoid unnecessary symbolic execution, we have the following online kill conditions that immediately terminate an offset's execution: exception due to an invalid instruction; invalid memory access such as a segmentation fault; any instruction does not align to the system call we know; detected system call number or address does not match given context from the IDS; execution of end of path system calls (e.g., `exit`, `exec`); and, jumps out of bounds of the memory buffer (we assume it contains the complete attack code).

Because any application level attack code must execute one or more segments of privileged code (i.e., system calls) to cause any real harm, we record the symbolically executed instructions that end with a system call as a *code fragment* for each starting offset. We leverage research from [5] and assert that valid fragments should have at least 6 instructions and contain at least 15 bytes.

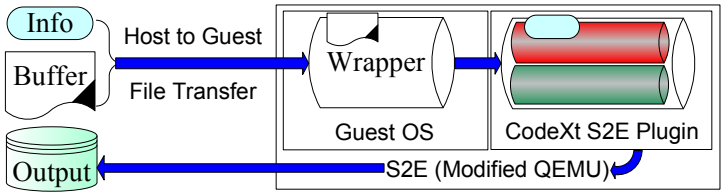

Fig. 4. Wrapper to Run Arbitrary Byte Stream in CodeXt

To model code with multiple system calls, we define a *code chunk* as a sequence of code fragments in a control flow. To extract code with multiple system calls, we merge adjacent code fragments into a *code chunk* if the following conditions are met: each code fragment itself is a code chunk; code chunks X and Y are adjacent if the start of Y is immediately after the logical end of X; and, if code chunks X and Y are adjacent and the end of X is not an end of path system call (e.g., `exit`, `exec`). This process may generate a number of code chunks, subsets of others are eliminated.

How to Handle Self-Modifying Code? In order to recover transient code involved in multiple layers of self-modification, we need to take snapshots for each layer of decoding. Since a defining characteristic of self-modifying code is executing dynamically generated instructions, we can reliably identify it if any instruction consists of bytes written by the code under observation. This can be achieved by tracking all the memory updates within the memory buffer range at run-time. However, we do not want to take a snapshot for each dynamically generated instruction as one layer of decoding normally consists of multiple correlated instruction blocks (e.g., `strcpy`). Instead we developed a clustering based approach for obtaining appropriate snapshots of self-modifying code.

We maintain a global counter of all the instructions executed, and we assign the current global counter to each to be executed instruction as its unique sequence number, which reflects the temporal order of the execution of all instructions. The memory updates within one layer of decoding tend to be clustered to each other in both time and space. Our heuristic clusters writes from memory update instructions whose execution sequence numbers are no more than Δ (e.g., $\Delta = 10$) apart and combines clusters with adjacent memory update ranges.

We treat one cluster of writes as one snapshot. We mark those snapshots from which we executed any instructions after the snapshot was created. These marked snapshots correspond to each layer of self-modifying code executed.

By stringing the snapshots together, we can generate a memory map to show the changes over time. Specifically, we can see all the values of all memory bytes translated, executed, or written, even if the same memory location has been overwritten multiple times during the execution.

Implementation We have implemented the prototype of CodeXt upon the S2E engine with 4,006 lines of C/C++ code. We have also extended the functionalities of S2E, KLEE and QEMU with 444 lines of C/C++ code. We have chosen to use the Strictly Consistent System-level Execution (SC-SE) consistency model

Surrounding Type	Run-time Hints	Code Found?	Sec. per Offset
Nulls	EIP, EAX	Yes	0.92
	EIP	Yes	0.94
	EAX	Yes	0.98
	Neither	Yes	0.98
Random	EIP, EAX	Yes	1.08
	EIP	Yes	1.09
	EAX	Yes	1.13
	Neither	Yes	1.11
Captured	EIP, EAX	Yes	1.04
	EIP	Yes	1.08
	EAX	Yes	1.00
	Neither	Yes	1.09

Table 1. Accuracy and speed for searching for the start of hidden code within a buffer.

that is "both strict and complete" [10]. Currently our prototype only monitors Linux system calls but it can be extended to monitor Windows system calls. Our prototype consists of a wrapper, shown in Figure 4, for loading an arbitrary byte stream for execution, a S2E plugin for online analysis, and a number of offline analysis modules. Our S2E plugin hooks into S2E's "CorePlugin" signals to define custom instructions and their handlers. It conducts deep analysis of the execution of any given byte stream from all offsets until a kill condition is reached, forwarding each offset's result to further offline analysis.

4 Empirical Evaluation

4.1 Locating the Hidden Code from Memory Dump

In this section, we evaluate CodeXt's capability in pinpointing the start and boundaries of potentially disjoint, misaligned code hidden in multiple execution paths from a given memory buffer. We vary two primary factors: the unrelated bytes surrounding the hidden code; and, amount of attack code context information (i.e., run-time hint). Specifically, we put our sample malware code into three types of buffers of different fill values for the bytes surrounding the code: 1) all nulls (`0x00`); 2) uniformly random; 3) capture (memory dump) from a real world code injection attack. Then we combined the three surrounding data types with varying amount of run-time hints: 1) with the address of a known system call (EIP) plus the system call number (EAX); 2) with the address of a known system call only; 3) with the system call number only; 4) neither.

Because it is easier to locate longer attack code, we deliberately used short attack code in our experiment: a 41-byte (per section 4.2) and 81-byte in-the-wild (ghttpd exploit) shell. We used a buffer of size 1024 bytes. We made the offset variable symbolic which directed CodeXt to explore 1024 potential paths.

For each of the four context information scenarios we experimented twenty runs with the random surrounding bytes, one run with fixed null surrounding bytes and one run with fixed captured bytes. CodeXt has successfully located the hidden code without any false positive in all runs for all the combinations.

Technique	Extracted?
Junk code insertion	Yes
Ranged XOR	Yes
Multi-layered	Yes
Incremental	Yes
ADMmutate	Yes
Clet	Yes
Alpha2	Yes
MSF Call+4 Dword XOR	Yes
MSF Single-byte XOR Countdown	Yes
MSF Variable-length Fnstenv/mov Dword XOR	Yes
MSF JMP/CALL XOR Additive Feedback Encoder	Yes
MSF BloXor	Yes
MSF Shikata-Ga-Nai	Yes

Table 2. Encoding Techniques Tested. MSF is Metasploit Framework. Multi-layered varied nesting combinations and depths of junk code insertion and ranged xor.

Table 1 shows the average time needed to search each offset for all the run combinations. It shows that the run-time hints do not have significant impact on the performance in any combination. It took about the same time (1 second per offset) to search through null, random and fixed captured surrounding bytes.

To validate CodeXt recovering multiple execution paths, we embedded the algorithm shown in Figure 1 into a 1KB buffer, and marked `x` to be symbolic. CodeXt successfully explored all the three feasible execution paths (it detected a fourth, `if (y==1 && z==0)`, as infeasible) and recovered their byte code.

We also investigated the probability of false positives. The probability for two consecutive random bytes to be the system call instruction `int 0x80` (`0xcd80`) is $2^{-16} = 1.52 \times 10^{-5}$. It is highly likely that a long string of random bytes contains some executable instructions, but far less likely to contain a false positive (i.e., set EAX to a specific value or range, and end in a system call before EAX is clobbered). In addition, previous research details that 90% of random strings should fail execution within 6 instructions [5]. We tested CodeXt with buffers of pure, uniformly random bytes of 1KB, 10KB and 100KB respectively. Specifically, we input 20 different 1KB, one 10KB and one 100KB random bytes. CodeXt did not reported any hidden code detected from these random bytes.

4.2 Extracting Encoded Bytes

To evaluate CodeXt extracting encoded bytes, we have used 12 different encoders to encode a running example of byte code, called hw.shell, that prints "Hello, world!" to the standard output via the `write` system call.

Besides using 9 well-known third party encoders (e.g., ADMmutate [1], Clet [11], Shikata-Ga-Nai [2]), we developed 3 encoders ourselves: the *junk code insertion* encoder and the *ranged xor* encoder based on [3], and a novel *incremental* encoder. Table 2 lists all 12 encoders plus an entry to represent multi-layer combinations (nested) of our in-house encoders. CodeXt was able to automatically recover the original shellcode in all tested cases. In the following subsections, we

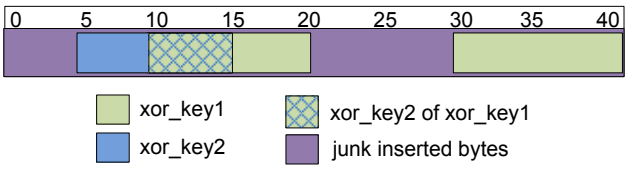

Fig. 5. Multiple layers of ranged xor encoding that overlap each other and use different keys, all layered on top of a junk code inserted encoding.

elaborate the experiments with multi-layer combinations of junk code insertion and ranged xor, the incremental encoder, and Shikata-Ga-Nai.

Multi-Layer Combinations To evaluate CodeXt's capability in extracting code protected with multiple layers of encoding, we tested combinations of our in-house encoders: junk code insertion and ranged xor. Junk code insertion interjects a random length of random value bytes between each input byte, such that $junk(i)$ means to generate encoded output from input i. Junk code insertion, while very rudimentary, effectively interferes with common disassemblers. Ranged xor uses a 1B key to iteratively encrypt a specified range of input bytes, such that $xor(k_n, o, b, i)$, means to encode input i with key k_n at offset o for b bytes; also we will use $xor(k_n, i)$ to mean encoding all bytes in i.

In our trials we tested combinations such as 1) xor of junk: $xor(k_1, junk(i))$, 2) junk of xor: $junk(xor(k_1, i))$, 3) xor of xor: $xor(k_2,\ xor(k_1, i))$, and, as illustrated in Figure 5, 4) xor of xof of xor of junk: $xor(k_2, 5, 10,\ xor(k_1, 30, 10,\ xor(k_1, 10, 10,\ junk(i))))$. CodeXt was able to recover the original shellcode from all tested multiple layer combinations of encodings.

Incremental Encoder We have developed a sophisticated incremental encoder, such that during decoding it will incrementally de-obfuscate one portion (or segment) of the original code at a time. After executing the decoded segment, it will decode another code segment into the same buffer and so on. Except the final decoded code segment, all other decoded segments are transient in that they will be overwritten right after execution. Therefore, a memory dump or snapshot at any moment will never reveal the entire decoded form. In order to extract the complete code protected by the incremental encoder, we need to take multiple snapshots at the right moments and places during run-time.

We have used the incremental encoder to encode a popular TCP based reverse connect shellcode with 5 system calls into 4 code segments, roughly representing its basic blocks. CodeXt generated 8 snapshots when executing the incrementally encoded shellcode. This accurately represents the algorithm copying each encoded segment into a buffer and then decoding it. CodeXt has successfully extracted the complete code protected by the incremental encoding.

Shikata-Ga-Nai Shikata-Ga-Nai is a polymorphic xor additive feedback encoder within the Metasploit Framework. This encoder offers three features that provide advanced protection when combined. First, the decoder stub generator uses metamorphic techniques, through code reordering and substitution, to

produce different output each time it is used, in an effort to avoid signature recognition. Second, it uses a chained self modifying key through additive feedback. This means that if the decoding input or keys are incorrect at any iteration then all subsequent output will be incorrect. Third, the decoder stub is itself partially obfuscated via self-modifying of the current basic block as well as armored against emulation using FPU instructions.

```
Offset Bytecode  Mnemonic             ; Comment
0000  DAD4       fcmovbe st4          ; any fpu insn
0002  B892BA1E5C mov eax,0x5c1eba92   ; key = 92ba1e5c
0007  D97424F4   fnstenv [esp-0xc]    ; write fpu records to
                                      ; put EIP on top of stack
000B  5B         pop ebx              ; ebx = EIP
000C  29C9       sub ecx,ecx          ; clear ecx
000E  B10B       mov cl,0xb           ; loop 11 times
0010  83C304     add ebx,byte +0x4    ; PC += 4
0013  314314     xor [ebx+0x14],eax   ; [0x0018] = [0x0018]^key
0016  034386     add eax,[ebx-0x7a]   ; key += [ebx + Encoded Byte]
0019  58         pop eax              ; False Instruction, Encoded Byte
001A  EBB7       jmp short 0xffffffd3 ; False Instruction, Encoded Bytes
```

The above code snippet shows the instructions from an instance of Metasploit's polymorphic xor additive feedback encoder Shikata-Ga-Nai [2]. It uses FPU instructions as a way to get EIP, via fnstenv/pop. Since most emulators, including S2E/QEMU, do not fully support FPU instructions, they have been used to detect emulated environments. For such FPU instruction combinations (e.g., fpstenv/pop), we implemented special handlers in our S2E plugin to emulate their semantics (e.g., update FPU internal values).

The instruction at address 0x0013 changes 4 bytes starting from address 0x0018, which changes the subsequent 3 instructions to be executed from address 0x0016. Therefore, Shikata-Ga-Nai dynamically modifies the instructions in the current basic block. S2E/QEMU was not able to handle such intra-basic block modification. To fix this problem, we extended the S2E translation mechanism so that our S2E plugin can force a re-translation should we detect any write within the memory range of upcoming instructions in the current basic block. With the added support of FPU instructions and intra-basic block modification, CodeXt has successfully extracted the hw.shell protected by Shikata-Ga-Nai.

5 Related Work

Attack Code Detection A number of methods [26, 9, 31, 20] have been proposed to detect attack code from network packet payloads based on various heuristics. Work [9] used static analysis based approach to look for the NOP sledge in the packet payload. SigFree [31] detects the presence of code from packets by checking the push-call instruction pattern and data flow anomaly from the static disassembled instruction sequences, and it depends on static disassembling. SHELLOS [24] utilizes hardware virtualization and KVM to detect the existence of injected code. Because it directly executes the instructions on the CPU, it can execute Shikata-Ga-Nai encodings that most CPU emulators (e.g., QEMU, S2E) can not. While existing attack code detection methods are able to detect the existence of attack code even if the attack code is mingled with

random data, they are not able to determine the exact location and boundary of the attack code. Therefore, existing attack code detection methods can not automatically extract the attack code.

Binary Code Extraction BCR [7] is not designed to extract the complete hidden code from a given memory dump, but rather to extract certain reusable code fragments from a given binary program. It requires the knowledge of the entry point to be effective and it can not handle self-modifying code.

Automated Unpacking of Hidden Code Automated unpacking hidden code [6, 34] has been an active research area and many methods [15, 21, 14, 19, 29, 4, 33, 16] have been proposed to address the unpacking issue. Earlier methods (e.g., [15]) have used static analysis, and later approaches have used combination of static and dynamic analysis. Notably, PolyUnpack [21] detects the self-modifying code by checking if the to-be-executed instruction sequence is part of the static code model generated before execution. Because of the need of static modeling, it is not easy to apply PolyUnpack to code packed with multiple layers. OmniUnpack [19] detects unpacking by looking for written-then-execute pattern. It ignores intermediate layers of unpacking and only takes actions upon the invocation of some dangerous system call, which is assumed to be after the innermost layer of unpacking. This is a similar with Justin [16] that assumes a known code entry point and takes action upon a singular confluence of events; which does not account for incremental unpacking. OmniUnpack operates at the granularity of memory page, and it does not give any information about the intermediate layers of unpacking. As a result, it is faster. Renovo [14] also uses the written-then-execute pattern to detect the unpacking. It checks at the granularity of basic block. Specifically, it dumps the memory pages that contain the current basic block and has been written recently. Eureka [25] is a coarse-grained unpacking approach that uses Windows specific heuristics and x86 code statistical pattern.

To the best of our knowledge, all existing automatic unpacking mechanisms require the knowledge of the exact start of the code, and they are not effective when the hidden code is mingled with other bytes (i.e., the exact start of the hidden code is unknown). In addition, most existing unpacking methods only recover the hidden code and data on one execution path. No existing generic unpacking approach has been shown to be able to handle Metasploit's polymorphic xor additive feedback encoder Shikata-Ga-Nai and the incremental encoder that encodes only a segment of the hidden code in each layer of encoding. In fact, SHELLOS [24] abandoned attempted QEMU-based approach because of QEMU's incapability in handling FPU instructions and Shikata-Ga-Nai.

In contrast, CodeXt is able to explore multiple execution paths via combination of symbolic execution and concrete execution and recover the hidden code and data on multiple execution paths. To the best of our knowledge, it is the first system that can automatically recover hidden code protected by Shikata-Ga-Nai and the incremental encoder that encodes only a segment of the hidden code in each layer of encoding.

6 Conclusion

Extracting attack code is indispensable for effective malware analysis, forensics and reverse engineering. No existing approach has been shown to be able to automatically recover 1) disjoint, misaligned attack code mingled with random bytes; and 2) those transient code protected by multi-layer incremental encoding.

In this paper, we have presented CodeXt to address the above mentioned challenges. Based on selective symbolic execution and its unique multi-layer snapshot, CodeXt is designed to accurately pinpoint the exact start and the boundaries of the attack code and recover the hidden and transient code protected by various multiple layers of self-modification. Our experiments with real world shellcode and shellcode encoders have demonstrated that CodeXt is able to accurately extract the hidden code mingled with random bytes even if the code is protected by sophisticated encoders such as polymorphic xor additive feedback mechanisms like Shikata-Ga-Nai. In addition, CodeXt is able to automatically recover the transient code protected by multi-later incremental encoding.

Acknowledgments This work was supported by NSF grant CNS-0845042.

References

1. ADMmutate Polymorphic Shellcode Engine. http://ktwo.ca/security.html.
2. Polymorphic XOR Additive Feedback Encoder. In the *Metasploit Framework*. http://metasploit.com/modules/encoder/x86/shikata_ga_nai.
3. Simple Obfuscation. http://funoverip.net/2011/09/simple-shellcode-obfuscation.
4. P. Bania. Generic Unpacking of Self-modifying, Aggressive, Packed Binary Programs. http://piotrbania.com/all/articles/pbania-dbi-unpacking2009.pdf.
5. G. Barrantes, D. Ackley, S. Forrest, and D. Stefanovic. Randomized Instruction Set Emulation. *ACM Trans. on Information Systems Security*, 8(1):3–40, 2005.
6. T. Broch and M. Morgenstern. Runtime Packers: The Hidden Problem? http://blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf.
7. J. Caballero, N. Johnson, S. McCamant, and D. Song. Binary Code Extraction and Interface Identification for Security Applications. In *Proc. of the 17th Netw. and Dist. System Security Symp.*, Feb 2010.
8. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th Symp. on Operating Systems Design and Implementation*, pages 209–224, Dec 2008.
9. R. Chinchani and E. van den Berg. A Fast Static Analysis Approach To Detect Exploit Code Inside Network Flows. In *Proc. of the 8th Int. Symp. on Recent Advances in Intrusion Detection*, Sep 2005.
10. V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proc. of the 16th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.
11. T. Detristan, T. Ulenspiegel, Y. Malcom, and M. von Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. In *Phrack*, issue 61, id 9, Aug 2003.
12. H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly Detection Using Call Stack Information. In *Proc. of the IEEE Symp. on Security and Privacy*, 2003.
13. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proc. of the IEEE Symp. on Security and Privacy*, 1996.
14. M. Kang and P. Yin. Renovo: a Hidden Code Extractor for Packed Executables. In *Proc. of the 2007 ACM Workshop on Recurring Malcode*, pages 46–53, 2007.

15. C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proc. of the 13th USENIX Security Symp.*, Aug 2004.
16. F. Guo, P. Ferrie, and T. Chiueh. A Study of the Packer Problem and its Solutions. In *Proc. of the 11th Int. Symp. on Recent Advances in Intrusion Detection*, 2008.
17. C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proc. of the 10th ACM Conf. on Computer and Commun. Security*, pages 272–280, Oct 2003.
18. C. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. Debray, and J. Hartman. Protecting against Unexpected System Calls. In *Proc. of the 14th USENIX Security Symp.*, Aug 2005.
19. L. Martignoni, M. Christodorescu, and S. Jha. Omniunpack: Fast, Generic, and Safe Unpacking of Malware. In *Proc. of the 23nd Annu. Computer Security Applications Conf.*, pages 431–441, 2007.
20. M. Polychronakis, K. Anagnostakis, and E. Markatos. Network-level Polymorphic Shellcode Detection Using Emulation. In *Proc. of the IEEE Conf. on Detection of Intrusions and Malware and Vulnerability Assessment*, pages 54–73, July 2006.
21. P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proc. of the 22nd Annu. Computer Security Applications Conf.*, 2006.
22. R. Sekar, M. Bendre, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proc. of the 2001 IEEE Symp. on Security and Privacy*, May 2001.
23. M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding Malware Analysis Using Conditional Code Obfuscation. In *Proc. of the 15th Network and Distributed System Security Symp.*, 2008.
24. K. Snow, S. Krishnan, F. Monrose, and N. Provos. SHELLOS: Enabling Fast Detection and Forensic Analysis of Code Injection Attacks. In *Proc. of the the 20th USENIX Security Symp.*, Aug 2011.
25. M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, and W. Lee. Eureka: A Framework for Enabling Static Malware Analysis. In *Proc. of the 13th European Symp. On Research In Computer Security*, pages 481–500, Oct 2008.
26. T. Toth and C. Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Proc. of the 5th Int. Symp. on Recent Advances in Intrusion Detection*, pages 274–291, 2002.
27. S. Udupa, S. Debray, and M. Madou. Deobfuscation: Reverse Engineering Obfuscated Code. In *Proc. of the 12th Working Conf. on Reverse Engineering*, 2005.
28. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proc. of the 2001 IEEE Symp. on Security and Privacy*, May 2001.
29. X. Wang, D. Feng, and P. Su. Reconstructing a Packed DLL Binary for Static Analysis. In *Proc. of the 5th Info. Security Practice and Experience Conf.*, 2009.
30. X. Wang and X. Jiang. Artificial Malware Immunization based on Dynamically Assigned Sense of Self. In *Proc. of the 13th Info. Security Conf.*, Oct 2010.
31. X. Wang, C. Pan, P. Liu, and S. Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proc. of the 15th USENIX Security Symp.*, Aug 2006.
32. C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions Using System Calls: Alternative Data Models. In *Proc. of IEEE Symp. Security Privacy*, 1999.
33. Y. Wu, T. Chiueh, and C. Zhao. Efficient and Automatic Instrumentation for Packed Binaries. In *ISA*, pages 307–316, 2009.
34. W. Yan, Z. Zhang, and N. Ansari. Revealing Packed Malware. *IEEE Security Privacy*, 6(5):65–69, Oct 2008.