Inspector Gadget: Automated Extraction of Proprietary Gadgets from Malware Binaries

Clemens Kolbitsch
Secure Systems Lab
Vienna University of Technology
Vienna, Austria
ck@iseclab.org

Thorsten Holz
Secure Systems Lab
Vienna University of Technology
Vienna, Austria
tho@iseclab.org

Christopher Kruegel
University of California
Santa Barbara, USA
chris@cs.ucsb.edu

Engin Kirda
Institute Eurecom
Sophia-Antipolis, France
kirda@eurecom.fr

***Abstract*—Unfortunately, malicious software is still an unsolved problem and a major threat on the Internet. An important component in the fight against malicious software is the analysis of malware samples: Only if an analyst understands the behavior of a given sample, she can design appropriate countermeasures. Manual approaches are frequently used to analyze certain key algorithms, such as downloading of encoded updates, or generating new DNS domains for command and control purposes.**

In this paper, we present a novel approach to *automatically* extract, from a given binary executable, the algorithm related to a certain activity of the sample. We isolate and extract these instructions and generate a so-called *gadget*, i.e., a stand-alone component that encapsulates a specific behavior. We make sure that a gadget can autonomously perform a specific task by including all relevant code and data into the gadget such that it can be executed in a self-contained fashion.

Gadgets are useful entities in analyzing malicious software: In particular, they are valuable for practitioners, as understanding a certain activity that is embedded in a binary sample (e.g., the update function) is still largely a manual and complex task. Our evaluation with several real-world samples demonstrates that our approach is versatile and useful in practice.

I. Introduction

Malicious software (*malware*) is the driving force behind many of the attacks on the Internet today. For example, spam e-mails are commonly sent via spambots, denial-of-service attacks caused by botnets threaten the availability of hosts on the Internet, and keyloggers steal confidential information from infected machines.

Although malware has been around for a long time, it has been significantly evolving in its nature. For example, whereas malware was largely distributed as individual, stand-alone programs ten years ago (e.g., viruses, worms), it is now being increasingly deployed as software that can be remotely controlled by its creators. Most malware instances implement some kind of communication channel between the running instance and the attacker. Typically, this channel is used to update, control, and communicate with malicious software. For example, the attacker can use the channel to send a malware instance new URLs that should be advertised via spam e-mails, new binary files that should be executed on the compromised host, or a list of targets for logging keystrokes. This remote configuration mechanism gives an attacker flexible control over the infected machine. Hence, she can arbitrarily configure the compromised host to carry out her malicious deeds.

Understanding what actions a given sample performs is important to be able to design corresponding countermeasures and mitigation techniques. For a security analyst, understanding the remote control mechanisms is especially interesting as these provide valuable clues about the malware. Unfortunately, analyzing the configuration mechanisms (and also all the other activities of a malware binary) is a challenging and complex task. Typically, the analyst does not have access to the source code of the malware sample. As a result, the analysis needs to operate on the binary executable. Furthermore, the analysis is complicated by the fact that the adversary can arm the binary with different kinds of obfuscation and evasion techniques (e.g., [1], [2]) to hamper and resist analysis. Thus, there is general consensus among practitioners that the static analysis of malware is generally a difficult task [3].

Because of the shortcomings of static techniques, dynamic analysis techniques are often used in practice. However, dynamic analysis also has some limitations (e.g., execution of a single path, identification of virtual environments, etc.) [4], [5]. Furthermore, such systems do not provide support for automatically extracting the configuration mechanism or other aspects of a sample under analysis.

In practice, a human analyst often needs to spend a considerable amount of time manually decoding and analyzing the malware sample in order to understand the key algorithms embedded in the sample. An example for such a key algorithm is the *domain generation algorithm* of malware samples that use *domain flux* [6]. With domain flux, each bot periodically generates a list of domains that are then used to contact the attacker. As the attacker knows the domain generation algorithm, she can set up an infrastructure and register these domains in advance. During the analysis, the analyst is interested in extracting these embedded algorithms

such that she can also precompute the domains that will be used in the future [7].

Another example of a key algorithm that needs to be manually analyzed is the decoding function that is embedded in a sample. The malware uses this function to decode obfuscated configuration files [8]. With the decoding function at hand, the analyst can decode and analyze spam templates that are sent to the malware.

In this paper, we aim at improving the state of the art by presenting a novel approach to *automatically* extract from a given malware binary the instructions that are responsible for a certain activity of the sample. We term these instructions a *gadget* since they encapsulate a specific behavior that can autonomously perform a particular task. The key idea behind our approach is that the malware binary itself has to contain all necessary instructions to perform the malicious operations that we are interested in. Hence, if we are able to isolate and extract these instructions (i.e., gadgets) in such a way that we can reuse them again in another application, we can perform a specific task of the malware (e.g., download the current set of URLs that should be advertised in spam mails) in a self-contained way, without the need of executing the whole malware binary. Note that we do not need to understand the behavior of the malware. We can simply *reuse* the code extracted from the sample.

To achieve this goal, we have implemented a tool called INSPECTOR (abbreviation for *Inspector Gadget*) that automatically extracts gadgets from a given malware binary. In a first phase, INSPECTOR performs dynamic program slicing [9] on the malware binary to extract a slice (i.e., an algorithm) with "interesting" behavior. This could be, for example, a slice that downloads a piece of binary data from the Internet, deobfuscates this data to obtain a binary executable, and then writes this file to the hard disk.

Clearly, applying program slicing to malicious input is a difficult task. However, we show in several case studies that INSPECTOR can indeed handle common obfuscation techniques such as binary packing or self-modifying code found in real-world malware. Note that we extract *complete algorithms* from the binary. This is more complex and difficult than only extracting specific functions (such as in [10]) since we need to consider all dependencies between functions, their side-effects, and relevant auxiliary instructions (e.g., stack manipulation, or loops).

In a second phase, INSPECTOR generates a *stand-alone* gadget based on the extracted algorithm. This gadget can then be executed to perform the specific task that was embedded into the malware binary. During the gadget generation process, we recursively include all intermediate code and additional data regions such as global variables into the gadget (i.e., *closure analysis*). All extracted memory regions are relocated such that we can later on execute the extracted code in another environment, the so called *gadget player*.

The gadgets we generate can perform all necessary actions that the original function embedded in the malware sample is to perform. That is, we do not need additional helper applications to relay the traffic between the extracted code and the network (e.g., such as network proxies as in [10]).

The case studies we used in our evaluation demonstrate that the gadgets we automatically generate provide the same malicious functionalities that were originally embedded into the malware samples. For example, we show that we can generate a gadget that autonomously downloads data from the network, and decodes it using a proprietary algorithm to obtain an executable. Another gadget we extracted enables us to decode encrypted network traffic. Furthermore, our transformation enables an analyst to influence the behavior of a given gadget by manipulating the function calls invoked by the extracted code. Using this feature, the analyst can perform a deeper analysis of the malicious functionality provided by the gadget. For example, she can intercept date checks, and return arbitrary values to the gadget to determine the effect on the execution.

In practice, executing extracted gadgets instead of the original malware has the following important advantages:

- Since we are dealing with malicious software, the sample is potentially harmful. If we can extract only the parts relevant to a certain computation and execute them in a stand-alone fashion, we reduce our exposure to the malicious code.
- We can immediately carry out a certain operation the malware performs, instead of requiring to wait for timeouts, sleep operations, or commands that are sent over the command and control server.
- We can identify in-memory buffers that hold decrypted data. These can be extracted easily with the help of the gadget compared to running the sample in a debugging environment, and manually inspecting memory.

Further, we also show how some gadgets can be *inverted*. That is, we can use a gadget as a black box to compute what specific input causes a given output. Inverting gadgets is useful in many real-world scenarios. For example, inversion can be invaluable for automatically decoding a network trace that was encoded by a specific malware sample under analysis. In this work, we show how INSPECTOR can use optimized brute-forcing techniques to compute these inverse gadgets, and demonstrate with the help of a practical example the usefulness of this technique.

In summary, we make the following contributions:

- We propose and implement a novel approach to enhance malware analysis. The core idea is to automatically extract self-contained, proprietary algorithms from a malware sample that can then be reused to execute the specific malicious functionality embedded in the sample.

- We introduce a technique to transform the extracted algorithm into a stand-alone executable (that we denote a *gadget*). This is a challenging task since we need to handle all dependencies (e.g., global variables and auxiliary instructions), and also relocate all code.
- We discuss how gadgets can be inverted. That is, we show how we can use a given gadget to compute the input for an observed output. This technique is useful, for example, for automatically decrypting an obfuscated network trace that the malware generates.
- To demonstrate the practical feasibility of our approach, we present several case studies with real-world malware samples from different families (e.g., spam bots, key-loggers, etc.). The experiments support our thesis that gadget code can be reused, while only requiring a very limited amount of manual analysis.

II. System Overview

In this section, we first briefly review the problem we are attacking, and provide a high-level overview of our approach.

A. Problem Definition

The problem of gadget extraction is defined as follows: Given a binary of a malicious sample and an interesting behavior that we have identified during its execution, we would like to extract this behavior as a stand-alone code fragment with all its instructions and data dependencies. Furthermore, when starting the code execution of this self-contained application, care needs to be taken to isolate the gadget from the rest of the system so that it cannot exhibit any unexpected and unforeseen malicious behavior (e.g., such as attacking the analysis environment).

B. System Overview

The gadget extraction process implemented by INSPECTOR consists of three consecutive phases: *Dynamic analysis*, *gadget extraction*, and *gadget playback*. The overall process is illustrated in Figure 1 in a schematic way.

In order to obtain an initial overview of the behavior exhibited by the malware sample, in a first step, we execute the sample in an analysis environment, specifically in a dynamic *analysis sandbox* [11]. This step provides us with a detailed overview of the actions performed by the sample. Besides logging all system activity such as network communication, file activities (such as created or modified files), and process interaction, the sandbox also performs detailed taint tracking analysis [12]. At the end of the dynamic analysis phase, we obtain a set of log files that contain all collected information. Using these log files as a starting point, we can then query the execution run for "interesting" behavior. In this work, we focus on configuration mechanisms of modern malware. Hence, a behavior that is interesting from our point of view would be the download and subsequent decoding of a malware binary. Also, the generation of domains that are relevant to the communication channel between the malware and the attacker would be worth analyzing. Besides the semi-manual, guided finding of starting points for the extraction of gadgets, we also implemented two heuristics to automatically identify these interesting behaviors (see Section III-C2 for details).

In a second step, our tool automatically *extracts* all the code responsible for the interesting behavior exhibited by the analyzed binary. The starting point of the extraction process is a *sink* that specifies when an interesting behavior has been observed. Commencing at this position, we perform backward binary program slicing and forward searching [9], [13]. For example, writing tainted data to the hard disk and then executing this data indicates an update process of the malware. The sink would be the creation of a new file. We would then search backwards for all instructions related to the creation of the file (including all network communication and the decoding process), and extract all related code and data. The collected taint information can be used to link all invocations of library or system calls that provide data that is propagated into the calls. These invocations, together with intermediate code and necessary data regions, are then extracted into the *gadget*.

In a third step, INSPECTOR provides a gadget player to enable a security analyst to execute the extracted gadgets. This program can be used to re-invoke the behavior, and it integrates the gadget into the running environment. Similar to per-process virtual machines [14], the gadget player serves as a thin layer between the gadget and the environment, providing dynamic data such as network input or file contents to the gadget. Thus, we can *reuse* the extracted behavior and execute specific tasks as if the complete malware binary is being run. For example, the extracted gadget can be used to contact the update server of a given malware sample, and download and decode the newest version of the original malicious code – without the need of executing the malware binary. The gadget player also applies strict policies for files the gadgets are allowed to access. It also has detailed logging capabilities on calls to the environment and memory.

To ease the management of generated gadgets, we maintain a *gadget repository*, in which all extracted gadgets are stored such that they can also be used later on.

III. Automated Extraction of Algorithms

In this section, we present details on finding and extracting behavior from running samples. First, we explain how we perform detailed analysis that provides INSPECTOR with the required log files, and allows a human analyst to select behavior that she is interested in. Second, we discuss the behavior extraction process. We use a running example throughout the rest of the paper to illustrate the details of our technique.

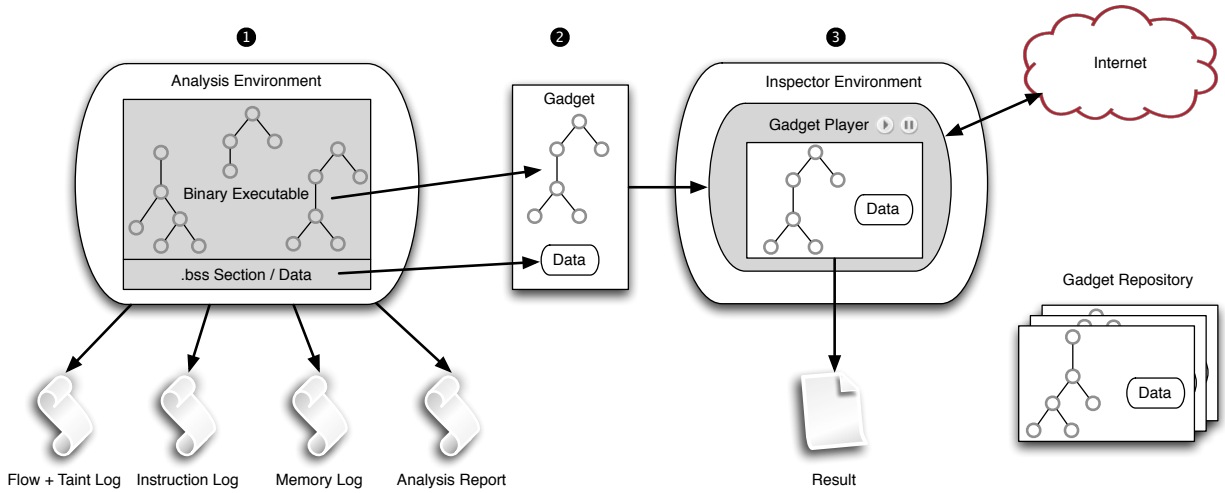


Figure 1. Schematic overview of gadget extraction process: (1) Execute the binary in analysis environment, perform dynamic analysis including taint tracking. (2) Extract *gadget* that represents specific behavior, intermediate code and additional data regions are extracted as well. (3) Gadgets can be autonomously executed within *gadget player* to perform a specific task.

A. Running Example

To illustrate the inner working of INSPECTOR, we explain the underlying concepts with the help of a running example, and the following settings: The malware binary we wish to analyze downloads a file from a static domain and a static URL. Since the downloaded file is encrypted, the malware first needs to decode the binary. Once this is finished, the downloaded file is executed. This is a popular *update mechanism*, a task often encountered when analyzing malware. We wish to extract from the given malware sample a gadget that encapsulates all these tasks in a stand-alone fashion. In later sections, we will extend this basic example as needed.

B. Generating Activity Logs

INSPECTOR first executes the malicious binary inside an analysis environment in order to gain an initial overview of the activities exhibited by the sample. Specifically, we use ANUBIS, a tool that performs dynamic malware analysis based on an instrumented processor emulator [15], [16]. The version of ANUBIS we use already implements some advanced features such as taint analysis, and the recording of all executed instructions [17]. We extended the tool to meet our specic analysis needs.

Besides concealing malicious behavior inside a safe environment, the analysis environment enables us to perform detailed taint analysis during the execution run. ANUBIS marks each byte returned by a system call with a unique label, and then keeps track of how labels are propagated during the program execution [12]. This enables us to observe how the input and output of different calls are related, and we are able to link individual computations. During the inversion of gadgets (see Section V), this detailed taint analysis is necessary to detect the interrelationship between all inputs of an output. The collected taint information is stored together with all arguments passed to and obtained from the operating system via library, or system calls in the log files.

Note that we also keep track of all disassembled instructions executed by the program, as well as the instruction flow (i.e., the sequence in which blocks of code and API functions are executed). These log files, which we call *instruction* and *flow log*, ease the later analysis of obfuscated malware. More precisely, we are able to analyze packed, or self-modifying binaries, reason about which conditional branches were executed, and keep track of statically undecidable call targets. To further aid the analysis process, we also record all memory accesses by the program and generate memory dumps of active pages during the program's execution.

A human analyst can use the generated analysis report to manually find the behavior she is interested in. For example, she may ask a question such as "What activity leads to the creation of this specific file?" Once she has spotted an interesting behavior in the analysis report, she can instruct INSPECTOR to extract a gadget related to this activity. The tool then analyzes all the collected log files and extracts from them the corresponding algorithm. Besides this semi-manual, guided extraction of gadgets, we also developed two heuristics to automatically locate activity that is worth investigating (see discussion in Section III-C2).

C. Selecting and Extracting Algorithms

As explained above, INSPECTOR can map a behavior selected by an analyst to a position inside the flow log (i.e., we denote this location *flow position*) of one of the monitored processes. Typically, a behavior directly corresponds to a system, or function call. However, it can also map to a set of instructions matching certain criteria (e.g., decoding of encoded data, as discussed later on). This

mapping is possible since we have all the relevant context information, and can find in the log files all activity related to the specified behavior. In case manual help is needed, INSPECTOR provides context information to the analyst in order to help her select the relevant flow. In the running example of the HTTP download, the flow position would be a call to one of the file management functions provided by Windows, such as `WriteFile`, or `CreateFile`.

Once we have identified the relevant flow position, our tool extracts a *slice*: It attempts to find all necessary data sources required to calculate the parameters passed to this function call, and extracts them. This is implemented by recursing on taint labels consumed by the API calls, as well as searching the memory logs for instructions defining variables (i.e., memory locations) read by the function invocation. We need to recursively identify all sources that somehow influence the parameters of the flow position since these instructions are potentially relevant to the algorithm we are about to analyze. We perform backward binary program slicing to compute this *closure*: We need to make sure that, during the extraction process, all relevant code and data is included recursively. When INSPECTOR finds a point which can be run in a self-contained fashion, this position is marked as the entry point and analysis ends. At the end of the process, the extracted code contains all information required to execute the specific behavior at the flow position in a stand-alone way.

1) Forward Searching and Backward Slicing: In some scenarios, the behavior selected by an analyst is not the intended endpoint of a chain of actions. Consider the case where the program downloads an *encrypted* list of URLs that are used to trigger keylogging activity. Here, extracting the download activity does not gain enough insight into the sought URLs since we are interested in the *decoded* list of URLs.

Thus, before extracting an algorithm, our tool allows the analyst to also *search forward* from a given position in the flow log (i.e., the initial flow position), filtering for function calls or instruction sequences that operate on data provided by the selected behavior. The analyst then needs to specify one or multiple function calls or blocks of code as an endpoints where the search stops. The analyst can also let INSPECTOR apply heuristics to determine the most likely endpoints. From there, we then perform backward slicing, and make sure that all relevant endpoints are also included in the closure such that the extracted algorithm is self-contained.

2) Heuristics for Detecting Endpoints: INSPECTOR uses the following heuristics to detect endpoints of interesting behavior. While calls to string comparison functions, or execution of code containing string handling instructions (such as `rep scas`) might not stand out particularly (or even be invisible when statically linked into an application), their occurrence is interesting when we perform forward searching. This is because it is an indication that the computations on the data that the instructions have touched have finished. In the case of encoded URLs, different string comparison functions might be used once the URLs have been decoded. Our tool can point out such behavior where data provided by the selected behavior is accessed. With this, an analyst can refine the selected flow position from the initial download to the point where the data is likely to have been decoded.

In fact, to support this frequent scenario of data decoding, INSPECTOR also provides the possibility to focus on data that has been processed by a list of mathematical instructions which indicate cryptographic activity [18]. Our empirical results indicate that this heuristic can reliably detect generic decoding functions, and select flow positions used as a starting point for the algorithm extraction.

3) Closure Analysis: In some situations, INSPECTOR can decide to deliberately exclude certain dependencies from the closure of extracted code based on two key observations.

First, we might simply not have all context information. This is due to the fact that the default application of INSPECTOR is to extract functionality from malicious binaries. We do not perform any static code extraction from the given sample, since this is, in general, a difficult task in the case of malware [1]–[3]. All analysis is based on log files generated during the dynamic analysis run (i.e., we can only reason about executed instructions and taken control paths). Whenever the backward slicing algorithm encounters a conditional jump whose alternative branch can redirect execution into a block containing code not executed during the dynamic analysis phase, the dependencies introduced by the condition are not added to the closure. Instead, the extracted algorithm is modified to include instructions that set the condition bits to force the branch to take the path that was observed during dynamic analysis.

Second, sometimes, a behavior is only triggered under a certain condition. In our running example, for instance, the update process might only occur on a special day of the week. Unfortunately, we can only include the specific behavior that we have seen during the dynamic analysis phase. At the same time, once we see a specific behavior, we wish to generate a gadget that *always* executes this behavior. Thus, INSPECTOR applies the same technique as discussed above to force the execution along a known path. We then include only this path, and skip the others. As a positive side effect, the program slicing is sometimes able to compute smaller closures, as whole blocks of functionality can be ignored during the extraction phase.

IV. GADGET PREPARATION AND REPLAY

In this section, we describe the process of encapsulating an extracted behavior so that it can later be re-invoked. Further, we introduce and outline the details of our system, the *gadget player*, that can be used to execute the gadget.

A. Gadget Format and Relocation

The format in which INSPECTOR stores the gadget code is a dynamically loadable library (DLL). This simplifies importing the behavior into the player for gadget execution (see next section). There are a number of ways in which we can bind and execute the gadget: On the one hand, we can use `LoadLibrary` to dynamically load a gadget. On the other hand, we can even statically link tools with the gadget.

To ensure that the gadget code runs in this relocatable fashion, all references to absolute code addresses (e.g., absolute call targets) are rewritten to use relative addressing. In contrast to static analysis, this step is simple as we know the complete execution flow of the extracted behavior.

While the library solves the problem of storing the gadget code, memory locations are not as straightforward. Such memory regions are typically *statically* allocated (i.e., the program expects them to be at a certain address). However, when re-executing the gadget, these memory locations might already be occupied by the system that is invoking the gadget.

Therefore, before exporting the code segment, all instructions are audited for static memory accesses: Using the memory logs, we decide if an instruction accesses a relocated memory area. If so, the disassembly of the instruction provides the immediate operands that might have to be patched. More precisely, we modify operands which are dereferenced directly or which are used as base-address for a memory access. Thus, we patch all operands so that they point to a location where we can ensure that the system will be able to allocate the memory for the gadget. Likewise, we parse the memory areas themselves for any static pointers that might point to relocated memory areas, and patch these appropriately.

As a last step, INSPECTOR extracts all static memory areas into a *data file*. For each area, it stores where the code will expect the data to be mapped and extracts the content from the ANUBIS memory snapshots at the point in execution where the extraction of the closure has finished. This corresponds to the entry point of the gadget. Therefore, when the gadget executes, all memory areas will contain exactly the same values as during the recorded execution.

Running Example: Using this slicing technique, INSPECTOR is able to extract a self-contained code fragment that performs the specific malware task. In our running example, the gadget includes all instructions relevant to calculating the URL of the file to be downloaded. Since the file is downloaded via HTTP, calls to `recv`, or similar functions related to network activity, are also included in the gadget. Furthermore, all the instructions related to decoding the downloaded content are also added.

At the end of the extraction process, the gadget contains all code and data related to actually performing the download, decoding the content, and saving the result to the hard disk. Note that we are not limited by function boundaries (as in [10]).

B. Gadget Player

To reuse a previously recorded behavior, INSPECTOR provides the so-called *gadget player*. The player's main functionality consists of three tasks: *Memory management*, *execution containment*, and *environment interface*.

1) Memory Management: From the gadget's data file, the player's *memory management* unit can identify all fixed, preinitialized memory areas that the gadget will rely on. Typically, such memory areas contain static strings and other global variables. These areas must be allocated and filled with the same values as were present in the analyzed process *before* the logged behavior started execution in ANUBIS. Additionally, the memory manager can also be called through the *environment interface* (see below) to handle dynamic memory allocation requests (e.g., through `RtlAllocateHeap`). Thus, while essential for proper execution of the gadgets, this unit additionally provides the player with a complete view of the memory buffers accessible to the gadget, and allows monitoring for changes made to them.

After initializing memory areas for the gadget, the player can load the behavior code, and start executing the gadget. As introduced in the previous section, a representation of the observed behavior is extracted into dynamically loadable libraries (DLLs). Thus, loading the behavior code can be easily achieved through the `LoadLibrary` function, which takes care of loading the code, and setting the appropriate permissions for it.

2) Execution Containment: When starting the actual code execution, special care must be taken to isolate the gadget from the player's memory, and handle possible crashes of the extracted code. Since the gadget is extracted automatically, we must make sure that we handle the execution robustly as there might be possible shortcomings of our extraction process. Also, because we deal with malicious code, special care must be taken in order to avoid undesired side effects. One possibility to guarantee isolation is by implementing *gadget emulation*. That is, the extracted code would be emulated. Because of performance considerations, we opted against this choice. In fact, emulation is not well-suited for tasks such as floating point operations that are known to be notoriously difficult to handle.

Our solution follows a different approach: As described in the previous section, most memory accesses have been statically rewritten to use the memory regions set up by the memory manager. Thus, the *execution containment unit* can natively run the gadget code, securing its execution inside a separate thread. For this thread, handlers for invalid memory accesses as well as execution of illegal instructions are registered to catch the most common source of errors. Further, during gadget extraction, we verify that the code is

free of any direct references to API or system calls. Thus, any kind of system interaction is forced to go through the *environment interface* (see below), allowing us to hinder the gadget from executing unintended malicious behavior. Last, the gadget's execution duration is limited to a configurable threshold in order to avoid deadlocks inside the extracted code.

A different approach [10] to contain the execution environment would be to implement software-based fault isolation (SFI) [19], [20]. Alternatively, one-way isolation [21] or similar techniques could be combined with concepts from NATIVE CLIENT [22] to contain execution. In the current prototype of INSPECTOR, the static rewriting of memory accesses is used. Our experience shows that it is a reliable and efficient way to contain the execution run within the player.

3) Environment Interface: The third component provided by the player is the *environment interface*. This component serves as the mediator between the gadget and the environment hosting the gadget player. During gadget start-up, the environment interface registers a callback function inside the gadget. This callback, implemented as a simple multiplexor function, is then invoked by the gadget each time a system or Windows API call would have been invoked during the malware execution.

Therefore, the environment interface must implement every kind of function that a gadget might request. This can be easily realized given the following key insight: By default, it is sufficient to redirect execution to the original library implementing the requested function instead of implementing functionality by hand. We can, thus, simply relay their implementation to an actual library. However, if an analyst wishes to manually interfere with the function call (e.g., to trigger a different behavior by returning a specific value), the environment interface also supports this. The analyst can manually implement a callback, which then performs the desired functionality. This can be especially useful in situations where the analyst decides to sanitize (or manipulate) data provided to or requested by the gadget.

During gadget extraction, INSPECTOR can verify that all required functions have been implemented in the environment interface, and inform the analyst about missing functionality. If the gadget player encounters a request for an unknown function (e.g., because the gadget was extracted by a newer version of INSPECTOR), it can decide to ignore the call, and continue execution. Obviously, this approach only works for functions that do not pop their arguments (i.e., use the `cdecl` x86 calling convention), since the stack layout might otherwise become corrupted.

4) Callback Handling: In the following, we describe two characteristic examples where we chose to implement functions inside the environment interface. In Microsoft Windows, the two functions `RegGetValue` and `RegQueryValueEx` provide means to retrieve the type and data for a specified registry value. Returning values provided by the hosting environment would be an acceptable solution here. However, intercepting the calls, and allowing the gadget player to return false information can uncover interesting information. Consider our running example of the update mechanism via HTTP: In this example, the download request could contain bits to indicate information about the host operating system version, allowing the attacker to provide different downloads, specifically targeted at the available host environment. By allowing the player to fake this information, INSPECTOR can easily trick the gadget into retrieving updates for a broad range of possible host environments. Therefore, a simple configuration option in the gadget player can save the analyst from having to re-run the gadget (or even the whole malware) in many different operating systems.

As a second example, consider Microsoft Windows's networking interface: In this example, the environment interface provides a wrapper for the actual networking implementation. Whenever calls to `connect`, `InternetConnect`, and related functions are encountered, the wrapper has the possibility to alter parameters before actually establishing a connection. Such parameters include the destination host and port. In our running example, this is particularly convenient for a security analyst in the case where the update binary is hosted on a fast-flux service network [23]. When running the download gadget repeatedly, it is very likely that the analyst will see different IP address in subsequent DNS lookups. Thus, each time, the request is served by different machines. Through a *configuration option*, the gadget player can be instrumented to always contact the same IP address, and allow to pinpoint the dates when a specific host starts serving a different, or updated binary.

Alternatively, instead of serving live network traffic, the network wrapper can also be instrumented to replay previously recorded network dumps (from `pcap` files, a format supported by many network analysis and recording tools). This technique enables interesting use cases from a forensic point of view: When provided with corresponding traffic dumps, the download gadget can extract binaries that were served at a different point in time. In cases where the network traffic contains dynamic data (such as keys used during the obfuscation process), we need to pay special attention. In the next section, we detail how this case can be handled by inverting gadgets.

V. GADGET INVERSION

Until now, the gadget discussion focused on cases in which malware samples interact with the environment or remote hosts (e.g., command and control servers). That is, the focus of the analysis was on what output information a gadget must produce so that the analyst can interact with a remote server. In practice, though, the inverse use case is also interesting.

Consider, for example, the case of information leakage due to a keylogger. Suppose that an analyst is given a network dump that contains information stolen by a keylogger where the data is encoded using a proprietary algorithm embedded into the malware. Furthermore, suppose that the analyst has a copy of the malware sample that is responsible for stealing the data. The task is now to find out what information was stolen (i.e., to determine what data was encoded by the malware and sent out over the network). To achieve this goal, the main idea is to first extract the gadget that is responsible for stealing and encoding the data. Second, we use the gadget and compute the input that leads to the output observed in the network dump. Thus, we would be able to determine what information was stolen in a reactive, forensic analysis setting.

In the following, we discuss how we realize this in practice. First, we need to change our perspective: In the previous sections, we treated the gadget as an object that invokes various library and system calls to interact with the operating environment and that translates (possibly altered) data in order to produce arbitrary output. In this section, we apply the same concepts as before. However, we simplify the gadget to a mere *transformation oracle* between input and output. This oracle can then be used to answer the question: "Using a given gadget, what output is generated if a certain input is provided?"

As we explained in Section II-B, INSPECTOR has complete knowledge over which sources provide data to which sinks based on the different types of log files we generate. This knowledge has byte granularity (i.e., for each byte in a sink, we can identify all input bytes that have an impact on the byte's value). We denote the relationship between different input bytes (sources) and an output byte (sink) a *source-sink dependency*. These dependencies can be used together with the gadget as an oracle. That is, we can brute-force, for each output byte, which input bytes need to be supplied to generate exactly this output. For a given output, we can, thus, determine what input leads to such an output. Hence, we can effectively invert the computation of the gadget transformation function. Although the brute-force approach we use to achieve this goal has some limitations in practice (see Section V-C), it is able to automatically deliver the expected results for certain kinds of gadgets.

To inverse a gadget, we use the following algorithm: Let $o \in O$ be the set of output bytes we are interested in and o_v the concrete value of the output byte o for which we seek to determine the input(s). Similar, $i \in I$ denotes the set of all input bytes transformed by the gadget. In a first step, using the source-sink dependencies, we find the set of dependent input bytes $D_o \subseteq I$ that have an influence on o, i.e., $\forall o \in O : D_o = \{i | i \in I \wedge o \text{ depends on } i\}$. Then, for each element in O, the set of *candidate* inputs C_o is determined by selecting all possible combinations of input values in D_o: $\forall o \in O : C_o = \{(v_{i_1} \times \ldots \times v_{i_n}) | (i_1, \ldots, i_n) = D_o \wedge v_i = value(i), v_i \in [0..255]\}$. Finally, for each candidate input $c \in C_o$, the gadget oracle is used to compute the candidate output o_c which can be compared to the desired value o_v. When an acceptable candidate c is found (i.e., both outputs o_v and o_c match), the process is repeated with the next element in O.

Special attention is required in the case where two or more output bytes $o, p \in O$ share dependent inputs, i.e., $D_o \cap D_p = D_{(o,p)} \neq \emptyset$. Here, each input candidate $c_p \in C_p$ can be discarded immediately if at least one input value $v_i \in c_p$ is assigned a different value than the same input byte in a previously accepted candidate $c_o \in C_o$. We call such inputs between o and p conflicting, or *dispute candidates*. Thus, the sequence in which output bytes are chosen must be done in such a way that those outputs containing less dispute candidates are selected first. If, at some point, no acceptable candidate for an output byte q can be found, the inversion algorithm must discard the previously accepted candidate $c_o \in C_o$, where $D_{(o,q)} \neq \emptyset$, and search for the next acceptable candidate c'_o (i.e, perform *backtracking*). If no other acceptable candidate can be found (and no dispute between o and another, previous element exists), the algorithm aborts with an error. Such a situation can occur, if there exists no input for the output O chosen by the analyst or INSPECTOR fails to find all source-sink dependencies for the selected behavior.

Otherwise, all output bytes are eventually assigned with an acceptable candidate input. By combining these candidates to a single set of input values $I_{accept} = \bigcup_{o \in O} \{c_o | c_o \in C_o \wedge o_v = oracle(c_o)\}$, we can thus answer the inverse question for a selected set of output values.

A crucial factor of this inversion strategy is that INSPECTOR assumes that the correspondence between input byte positions and output byte positions will remain constant as input changes. This implies that the algorithm does not handle optional or variable-length tokens in the input.

A. Inversion Example

To explain our approach in more detail, we revisit the example introduced at the beginning of this section. A keylogger steals sensitive data from a compromised host, encodes it using a proprietary algorithm, and then sends it out over the network to the command and control server. As an analyst, we only have access to the encoded network traffic and a copy of the keylogger. The goal is now to find out what data has been stolen by the malware.

Based on the network traffic, we know all the expected values of the output O. Next, we need to identify the corresponding sources D_o influencing the output. When recording the behavior of the keylogger in the analysis environment, INSPECTOR will identify a number of source-sink dependencies between the input bytes (e.g., stolen information read from the browser process) and the output bytes (encoded data sent over the network) since they are related to

each other due to taint analysis. Based on this information, we can compute how each output byte depends on the input bytes. Once we have identified the dependent inputs, we can compute the candidate inputs and use them together with the gadget: For each candidate input $c \in C_o$, we test with the help of the gadget what output is generated for this particular input. If the output matches the expected value, we have identified an input (i.e., a piece of information that was stolen). By repeating this process, we can recover, step by step, the complete input that was recorded by the keylogger.

B. Implementation Details

Inverting gadgets consists of two basic tasks: Extracting source-sink dependencies and evaluating input candidates.

1) Extracting Dependencies: During the dynamic analysis phase, we perform detailed taint tracking and record all dependencies between two labels. This enables us now to keep track of how a given sink is influenced by sources (i.e., what input bytes have an impact on a given output byte). The output of this first task is a mapping between source and sink bytes of a gadget.

2) Evaluating Input Candidates: The task of finding acceptable input candidates is implemented using a small helper application, the *brute-forcer*, and an extension to the gadget player.

Using the mapping file from the first step, the *brute-forcer* implements the generation of the set of input candidates as explained earlier. Additionally, it determines the sequence in which the output bytes will be checked. Then, starting with the first output byte, each possible input candidate is evaluated by calling the gadget player.

The gadget player is extended as follows: Each call to the environment interface can be handled by a chain of optional *interceptor components*. Each interceptor has the ability to inspect incoming function calls, and modify outgoing function arguments. It can also decide to call the next interceptor (or if there is none, the environment interface). Our implementation of the brute-force component is based on such an interceptor: On start-up, the brute-forcer initializes the interceptor with the current candidate inputs to provide to the gadget, and it registers all values it should check on incoming call arguments. During gadget execution, the interceptor keeps track of all function calls made by the gadget, and checks for input or output parameters in its list of monitored arguments. For each monitored outgoing parameter, the parameter value is overwritten with the provided candidate value. If the interceptor finds a monitored incoming argument, it examines the parameter's content. If a mismatch to the expected parameter is found, execution aborts and signals an error. Otherwise, call handling is delegated to the next interceptor. Once the interceptor has successfully verified all provided input arguments, it exits signaling successful execution.

With this extension, the extracted gadgets and the gadget player can be used in the inversion process without any modification. Once the interceptor has signaled successful execution to the brute-forcer for all output bytes, we have successfully inverted a gadget computation.

C. Inversion Applicability

A critical factor of our gadget inversion – as in every brute-forcing system – is related to the size of the input candidate set. If the number of candidate inputs $|C_o|$ that must be evaluated for a certain output byte o is very large, the time to identify an appropriate input set quickly becomes unmanageable. Likewise, the time necessary for finding an acceptable candidate for each output byte grows exponentially with the number of shared dependent inputs.

In order to assess the feasibility of inversion in different scenarios, consider the following three examples:

Base64 Encoding: In Base64 encoding (i.e., a specific type of MIME encoding), the set of input bytes is transformed into a base64 representation. According to RFC 2045, this is computed as follows: "A 24-bit input group is formed by concatenating 3 8-bit input groups. These 24 bits are then treated as 4 concatenated 6-bit groups, each of which is translated into a single digit in the base64 alphabet." [24].

The input for each step consist of three bytes (= 24 bits). For the computation, this is split into smaller parts of six bits each. Therefore, the set of input bytes is at most two $i_{1,2}$, which is then transformed into one byte of printable output o_1. With $|C_o| = 65536$ and $max(|D_{(o,p)}|) = 1, \forall (o,p) \in O$, gadget inversion is trivially possible.

XOR Encryption: When using XOR encryption, the computation is rather simple: One byte of input is xor-ed with one byte of the key to obtain one byte of output. Given that the key is known (e.g., it is statically encoded in the gadget), $|C_o| = 256$ and $D_{(o,p)} = \emptyset, \forall (o,p) \in O$, gadget inversion is even simpler than in the Base64 encoding example. If the key is part of the inversion input (e.g., it is part of the data received over the network) the algorithm will start with $max(|D_{(o,p)}|) = 1, \forall (o,p) \in O$ and $|C_o| = 65536$. However, after the first n candidates have been found (where n is the size of the key), the dispute candidates decrease the size of the input set to the previous case. This is because the algorithm has "found" the key to use.

Strong Encryption: For gadgets relying on *strong* encryption schemes such as RSA, the inversion fails with growing sizes of the output n. Since every output byte depends on all bytes of the key and input, $|C_o| = 256^n$ and $|D_{(o,p)}| \approx n, \forall (o,p) \in O$ make inversion impossible for large n.

Based on these examples, we can see that in the general case the following holds for the set of candidate inputs:

$$|C_o| \leq 256^{max(|D_{(o,p)}|)+1}, \forall p \in O$$

This indicates that we can only perform brute-forcing if and only if $|D_{(o,p)}|$ is small since else the computational effort to try all possible inputs quickly becomes intractable.

D. Possible Extensions

We have integrated the brute-forcing approach into INSPECTOR and can use it to invert certain computations of gadgets as we explain in the next section. In the future, we plan to improve the current approach as follows.

In addition to the source-sink dependencies, INSPECTOR also knows all instructions that modify the source data into sinks. Thus, we can perform symbolic execution to limit the search space, or we can – for simple transformation algorithms – even extract algebraic formulae. These formulae could be analyzed with a constraint solver to circumvent the costly input brute-forcing. However, brute-forcing is general and can also be used for encoding and encryption algorithms that cannot be stated as a compact formula, we opted against the solver approach. Moreover, as INSPECTOR can identify all dependencies that can be solved easily, a hybrid approach, combining a constraint solver and the brute-forcing approach, could yield better results. In this hybrid model, INSPECTOR could first solve all possible inputs, minimizing the number of parameters that need to be guessed.

Another possible improvement of the current approach would be through *input parallelization*: We can combine the set of independent output bytes (i.e., elements that do not contain common source-sink dependencies) and check multiple input candidates within a single gadget invocation. In practice, this improvement applies to many real-world scenarios: Most encoding, as well as simple obfuscation gadgets could be inverted significantly faster.

VI. EVALUATION

In order to demonstrate the feasibility of our approach, we generated gadgets in six case studies that involved well-known real-world malware from four different families. Our experiments show that we can reliably extract gadgets from a variety of samples in versatile ways. We chose these case studies because they cover the typical tasks that a malware security analyst would be interested in.

Table I summarizes various properties of the extracted gadgets. In particular, we can see that all but one executable used for the evaluation were packed. This shows that INSPECTOR can indeed handle state-of-the-art, obfuscated malware samples. Further, one can see that the gadget extraction was able to extract rather concise code snippets, eliminating most of the original executable's instructions during the closure analysis.

A. Domain Flux: Conficker

Bot families such as Torpig [6] and Conficker [25] employ the technique of *domain flux* to hinder the tracking of the communication channel between a bot and the attacker. With domain flux, using a proprietary algorithm, each bot instance periodically generates a list of domains that are used for obtaining commands from the attacker. The bot then contacts a subset of these domains until it finds an active domain (that has been registered by the attacker) from which it can receive commands. We are interested in extracting the *domain generation algorithm* (DGA) such that we can compute the set of domains used by a bot on a given date.

In our experiments, we studied Conficker.A since it employs the technique of domain flux to regularly generate a new set of domains that are contacted by the malware binary for updates. The malware implements the algorithm shown in Figure 2a in order to generate 250 domains. Note that all known details about the DGA were published in a paper by Porras et al. [25], who had to manually analyze the sample. There exist two parts of the report that are relevant for our analysis. First, Conficker contacts a remote HTTP server in the function `get_date_from_url` to obtain the current timestamp. Hence, an analyst cannot modify the local clock to trick the binary into generating domains for a particular date. Whereas an analyst could still change the timestamp in the HTTP reply, in the future, such an approach could potentially be complicated by the malware by switching to an SSL-based protocol instead of a cleartext one. Second, the function to actually generate domains contains floating point operations (that are supported by INSPECTOR).

In order to analyze Conficker, we first execute the sample within our analysis environment. After sleeping for 30 minutes, Conficker starts the DGA, and once the algorithm has been completed, it begins to resolve domains to also contact remote servers.

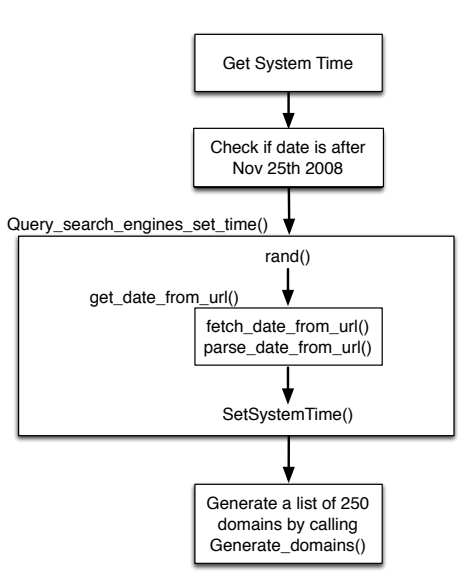
At this point, we can stop the execution, and begin to automatically extract the gadget. Since we are interested in the DNS activity, the flow position with which we start is a call to the function `gethostbyname`. From there on, INSPECTOR performs backward slicing based on all the collected log files, and identifies the code related to this function call. The tool recursively examines all code locations which influence the chosen flow position, and extract all relevant code together with the necessary data.

The output of the extraction and preparation process is a fully-functional gadget. The gadget includes all code related to the DGA, and we depict the data dependency graph in Figure 2b. The automatically extracted algorithm closely matches the manual analysis results shown in Figure 2a. Note that our gadget does not include a date check: Since the alternative path was not taken during the dynamic analysis phase, it is excluded in the preparation phase. The gray boxes depict taint information, where the bold text indicates that this input influences the actual computation of the DGA.

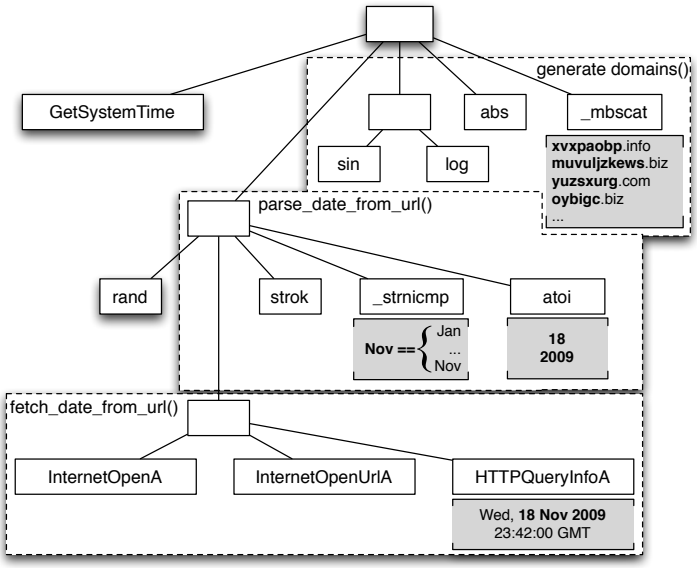
Note that we do not need to understand the algorithm: We can simply treat it as a black box to generate the current set of flux domains used by Conficker. When executing

Table I
OVERVIEW OF GADGETS EXTRACTED BY INSPECTOR.

Sample	Gadget	# Instructions extracted[1]	# Functions extracted	# API function references	Contains dynamically unpacked code
Conficker	Domain Flux	385 (511)	8	23	*yes*
Pushdo	Binary Update	926 (1410)	15	19	*no*
Cutwail	Spam Template	2091 (3575)	51	19	*yes*
URLZone	Configuration	1036 (1430)	27	17	*yes*



(a) Logic behind Conficker's domain generation algorithm (based on manual analysis [25]).



(b) Excerpt from Conficker's data dependency graph for domain generation algorithm (automatically generated by INSPECTOR, annotated for presentation).

Figure 2. Analysis results for Conficker's domain generation algorithm.

the extracted gadget in the player, the gadget outputs the current set of 250 domains. An additional advantage of our gadget is that it does not need to sleep for 30 minutes before starting the DGA. In contrast, it immediately begins with the computation and outputs the results.

To verify that our gadget correctly generates domains, we compared its output with the output of a human-generated tool that is based on manual analysis of the binary [7]. In all tests on 16 different days, our tool correctly computed the set of domains.

The gadget also enables an analyst to compute a set of domains for an arbitrary date, even one that lies in the future. As shown in Figure 2b in bold, INSPECTOR can extract which bytes are relevant for the computation of the DGA. In this specific case study, these are nine bytes related to a timestamp that are extracted from the HTTP response. Based on this information, an analyst can then implement a callback or an interceptor component in the environment interface that returns a different timestamp. As a result, the gadget performs the DGA for a different timestamp, effectively computing the set of flux domains for an arbitrary date. The callback interface in the environment interface, hence, eases the analyst's job and the analyst is able to adjust the gadget to her requirements.

B. Fetching Binary Updates: Pushdo

A common task that is implemented by malware instances is an *update mechanism*. That is, the malware downloads an arbitrary binary executable from the network, decodes it, and then executes this file. In fact, we have used this common mechanism as our running example throughout the paper.

In this case, we wish to recover the *decoded* binary executable. Therefore, we extract from the given malware binary a gadget that can perform the downloading and decoding steps in a self-contained manner. A good example for malware with this behavior is Pushdo. This sophisticated malware is capable of downloading additional components onto an infected machine, while hiding the attacker's traces [26].

After starting the sample in ANUBIS, we can observe several HTTP packets between the analysis environment and

[1]To facilitate handling of jump targets, the current implementation of INSPECTOR tries to preserve a function's structure whenever possible. Thus, instructions excluded from a gadget's code body are replaced by no-operation instructions (NOPs). Table I shows number of non-NOP instructions (number of *all* instructions are given in brackets).

a remote server. This activity is followed by file creation, and then the execution of the created files. At this point, we stop the analysis and extract a gadget that implements the file decoding and creation. Starting with the content that is written to a file, INSPECTOR detects the dependencies to the input received in the HTTP reply. Thus, the gadget includes the entire HTTP conversation, together with the code for building the appropriate request.

During execution, the gadget queries the environment for various system properties by reading registry values and performing low-level file system interactions. For our experiments, we allowed all read accesses to the host environment, ignoring any changes requested by the gadget code. After this initialization phase, the gadget starts to contact a remote server. Once successfully connected, it downloads binary data using the standard HTTP protocol, then transforms this data, and writes the result to a file.

Over a period of 16 days, we used this gadget to actively monitor binary updates served by three different command and control servers observed in recent ANUBIS submissions. While the extracted gadget always tries to contact the same IP for updates, we used a configuration option in the player to modify the contacted host. The results show that each server delivered a different executable (measured by the MD5 checksum of the decrypted binary). However, the served files per server did not change over the monitoring period.

C. Binary Update Decryption: Pushdo

In the previous case study, we demonstrated how we are able to actively download and decode a binary from a live command and control server. In some situations, however, it would be convenient for an analyst to have the possibility to *passively* decrypt recorded, or live network traffic generated by a machine infected with Pushdo. An analyst can then examine (in a forensic setting) the binaries downloaded by a specific host.

Unfortunately, the nature of the update protocol used by Pushdo creates some challenges for the analyst. Instead of downloading a *static* URL during the update process, a Pushdo client first generates a random sequence of bytes to be used as decryption key. This key is then *encoded* and appended to the static part of the URL. In turn, the command and control server splits the URL into static and key parts, decodes the key, and uses it to encrypt the file sent to the requesting client [26].

Within the network dump, we can, thus, only observe the encoded key. Therefore, we need to invert the Pushdo gadget from the previous example to obtain the decoded key. As the target of the brute-forcing process, we specify the outgoing HTTP request. INSPECTOR then automatically identifies the sources from the random number generator, and searches for acceptable input values until the generated request matches the one found inside the recorded network traffic. Once the requests match, we have found the key to use for the decryption. With this information, we can then analyze the HTTP reply and decrypt its content.

We have tested the decryption on ten different network dumps and were able to successfully extract the downloaded binary in all cases. On average, the inversion process finished after less than 40 seconds. This demonstrates the effectiveness and usefulness of the gadget inversion on real malware behavior.

D. Binary Update Generation: Pushdo

To further evaluate the capabilities of the gadget inversion, we extended the previous use case as follows: After extracting the decryption key, we specified the binary the gadget should write to file. That is, since the file content depends on data received from the update server, we allowed the brute-forcer to manipulate the bytes received from the network.

The practical use case for this problem is the following: An administrator can redirect binary update requests from a machine infected with Pushdo to a local HTTP server. This HTTP server uses our gadget inversion technique to generate an encrypted binary that uses, as key, the encoded key received within the request. The pushed binary (which could actually be a disinfection tool in this case) is subsequently downloaded, decoded, and executed by the requesting host.

In principle, the same concepts as in the pcap decoding example can be applied to this use case. However, we have to consider an additional difficulty: Even for small applications, a typical Windows binary is much larger than several kilobytes in size. Therefore, brute-forcing a complete binary within a reasonable amount of time is infeasible. Our solution to this is simple: We encrypt a minimal helper application that contacts our HTTP server and downloads an (unencrypted) DLL that contains the actual payload.

For our tests, we used *TinyPE* [27], a binary of 140 bytes in size, and a simple HTTP server written in Python. On a MacBook Pro with a 2.8 GHz Intel Core 2 Duo CPU, we were able to generate an encrypted binary within 444 seconds on average. However, an interesting property allows us to significant improve these results: Using INSPECTOR, we can see that, similar to the XOR encryption mentioned in Section V-C, each of the four key bytes is used independently for decrypting one fourth of the network input. This allows us to split the 2^{32} possible keys/requests into $4 * 256$ independent, encrypted inputs. Upon receiving an HTTP request, these inputs can be combined to form a single, valid reply. On the same machine, we can, thus, precompute all possible replies in well below 1.5 days.

E. Template-based Spamming: Cutwail

Current spambots typically use *template-based spamming*, a specific technique of sending spam in which the attacker sends each bot a *spam template* that describes the structure of the spam message to be sent [28]. In addition, the bot

also receives additional meta-data (e.g., recipient list or a list of URLs) that is then used to generate and send new spam mails.

In this use case, we are interested in extracting a gadget that performs the proprietary communication between a host and the command and control server, together with all relevant decoding steps. Such a gadget enables us to obtain the spam template, and we can observe what spam mails a bot is supposed to send out currently. This allows us to track the botnet, and we can use the collected information to significantly improve existing systems such as AutoRE [29] and Botlab [30]. These systems rely on executing a copy of the bot and collecting spam mails that are sent out. They reconstruct the actual template from this collected data. Using our gadget extraction approach, we can immediately obtain the full template, and do not need to reconstruct it based on network traces and the running of a (potentially dangerous) copy of the malware sample.

A bot that is commonly seen in connection with Pushdo is Cutwail [26]. This malware family is often downloaded by machines infected with Pushdo via the update mechanism, and it is responsible for sending out spam mails. The downloading of the templates, and the entire communication between an infected machine and the command and control server is encoded using a proprietary algorithm. A key of length n is used that is embedded in the binary [26]. The algorithm that the malware uses is the following:

1) Cutwail divides the encrypted string into blocks of length that equal to the length of the current key.
2) Each block is then XORed with the key.
3) The result is reversed (byte 1 and n are swapped, 2 and $n-1$, etc.).
4) Even-numbered blocks (e.g. Block 2, 4, ...) are also NOTed.
5) Finally, the remaining bytes which do not fit into a full sized block are simply NOTed.

The communication is handled in three stages. First, Cutwail downloads the current configuration settings, which includes information such as the connection timeout, the maximum numbers of attempts to send out mails, and the delays that the malware should respect. Second, a handshake is performed. Third, the malware downloads the spam template together with all meta-information such as target e-mail addresses. Once this data is decoded using the above algorithm, Cutwail starts to send out mails.

During the analysis of Cutwail, in order to extract the gadget, we first execute a Cutwail sample in our analysis environment. Once the bot starts to send out spam mails, we can terminate the dynamic analysis step since we can be sure that all relevant communication has already taken place. We select all calls receiving the encrypted content as our initial flow positions. INSPECTOR then extracts the relevant algorithm, and generates a stand-alone gadget that executes the same operations.

```
"{_FIRSTNAME} {_LASTNAME}" <{MAIL_FROM}>

Hello my new friend, I search a good man at other
country...\n For me it to communicate for the first
time with the person from other country, by
Internet.\nAnd it
...

{nReceived}
Message-ID: <{DIGIT[10]}.{SYMBOL[8]} {DIGIT[6]} @{nHOST}>
From:        {TAGMAILFROM}
To:         <{MAIL_TO}>
Subject:     {SUBJECT}
Date:        {DATE}
MIME-Version: 1.0
Content-Type: multipart/mixed;
boundary="----=_NextPart_000_0006_{_nOutlook_Boundary}"
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: Microsoft Outlook Express {_nOutlookExpress_4}
```

Figure 3. Excerpt from spam template extracted by Cutwail C&C communication gadget.

Upon invocation of the extracted gadget, we obtain information from the same command and control server that was contacted during the dynamic analysis run. This is because the IP address to contact is hard-coded inside the binary. We can, again, use the configuration options of the gadget player to modify this IP address, enabling us to monitor multiple command and control servers concurrently. Figure 3 shows an excerpt of a decoded spam template. The configuration options are simple key-value pairs such as `knockdelay 60` or `maxtryconn 5`. Note that the spam engine never stores any of this decoded information in a file, but keeps all information only in memory. Thus, this information cannot be obtained by simply executing the spam engine. An analyst needs to manually analyze the operations with a debugger, or by some other means.

F. Configuration of Keylogger: URLZone

Modern keyloggers enable an attacker to specify which websites should be monitored on the machine of a victim [31]. Each time the victim accesses one of these sites, the keylogger starts to record the information that the attacker is interested in. For example, the attacker is often interested in username and password combinations, or similar sensitive data. The dynamic configuration mechanism is usually implemented by downloading a configuration file from the command and control server right after the keylogger has started. The configuration file is commonly encoded using a proprietary algorithm. Thus, the malware first decodes the file, and then starts to monitor the activities of the victim.

In this case study, we are interested in extracting a gadget that contains the instructions related to downloading and decoding the configuration file. We can then monitor the current configuration of a keylogger, and learn which websites are interesting for an attacker. Since we can periodically execute the gadget, we can also continuously observe the

```
======================POST======================
...
[ITBEGINBLOCKHOOK]
ITHOST=|banking.postbank.de|End
ITPAGE=|/app/login.d*|End
ITMETHOD=|2|End
ITIFINIT=|%DISP%|End
ITREQMATH=|jsOn=*&accountNumber=*&pinNumber=*|End
...
---------------------- STATA ---------------------
ITINJHOST=|my.hypovereinsbank.de|End
ITINJPAGE=|/*?view=/*|End
...
ITINJPASTE=|%HYPOBAL%+%AMOUNT%-%TRUEAMOUNT%|End
ITINJPASTEMN=|<span class="negative-balance">
     %HYPOBAL%+%AMOUNT%-%TRUEAMOUNT%</span>
  <span class="negative-balance">EUR</span>|End
```

Figure 4. Configuration options revealed by URLZone download gadget.

activities of an attacker, and detect changes in the attacked target websites.

URLZone is one of the most advanced keyloggers currently found in the wild. Besides common functionality found in modern keyloggers such as the ability to collect user credentials entered by the victim or the ability to inject HTML code into web pages, this malware can also perform man-in-the-middle attacks against banking applications [32]. The malware downloads an encoded configuration file from the command and control server that specifies which URLs should be monitored for credentials.

Similar to the previous case studies, our goal is to extract a gadget that enables us to obtain the current configuration file from the command and control server in a decoded format. Hence, we start by executing a sample of URLZone in our analysis environment. Once the malware has downloaded the current configuration file, we stop the execution, and begin with the gadget extraction process. Similar to the Pushdo gadget above, based on the download activity, we let INSPECTOR find an appropriate target flow in order to start the gadget extraction.

In experiments, using the gadget, we monitored one active command and control server over a period of eleven days by invoking the gadget on an hourly basis (note that this server is still active at the time of writing this paper). On each invocation, the gadget successfully extracted a configuration file, as well as templates for altering displayed webpages to conceal its information stealing attack [32]. Extracts from this data can be seen in Figure 4. Interestingly, all downloads provided the same decrypted content. This could be because URLZone has been seen in the wild for quite some time. Thus, the current templates could have proven to be reliable. Also, the monitoring time might not have been long enough.

VII. RELATED WORK

Given the importance and the threat that malicious code poses, it is not surprising that there has been a significant amount of work on malware analysis and detection, both using static and dynamic techniques (e.g., [33]–[36]). Also, binary program slicing [9], [13], [37] and taint analysis [12] are standard techniques that are frequently used. Compared to previous approaches, our novel contribution is the automated extraction of proprietary algorithms that are embedded in malware. That is, we reuse existing code and transform it into a stand-alone gadget that can be used to (re)execute specific malware functionality.

Concurrently and independently of our work, Caballero et al. proposed BCR [10]. BCR is a tool that aims to extract a function from a (malware) binary so that it can be reused later. Compared to BCR, INSPECTOR has the following advantages: First, BCR is only able to extract a *single function*, while we extract the entire *functionality* from a binary. Finding a particular, interesting function (and its entry point) is a difficult task in itself. We do not have to solve this problem, since our techniques extract the entire algorithm that translates program inputs (via system calls) to program outputs. Of course, such external inputs and outputs are much easier to identify than internal functions. Second, we embed an extracted algorithm automatically into a stand-alone component (a gadget) that can be used by an analyst to "replay" malware actions. This is different from BCR, where the analyst has to manually develop additional code that makes use of the extracted functionality. As an example, with BCR, an extracted encryption routine would need to be embedded into a network proxy to be able to download and decrypt a binary update. In our case, INSPECTOR will generate a gadget that automates the complete process of downloading and decrypting this binary update. Third, we provide a mechanism to invert the functionality of an algorithm so that we can find the inputs that lead to certain outputs. This is valuable when an analyst wants to decrypt/decode data that was previously encrypted/encoded by a malware sample.

Lin et al. introduced an attack that extracts an interface to functionality in a benign program to add malicious functionality [38]. The idea is to re-use existing code within a binary (in a sense similar to return-oriented programming [39], [40]) and transform the binary such that malicious activities are performed (e.g., turning an e-mail client into a spam-sending trojan). The general concept of reusing binary functionality is related. However, we are interested in isolating the algorithm from a given (malicious) binary that is responsible for a certain activity.

Our approach could be seen as an extension to the problem of protocol dialog replay [41], [42]. However, while these approaches only inspect the network-level aspects of malware communication (between the malware program and its command and control server), we also include the host-level operations. For example, we can decrypt an encoded binary that was downloaded from a remote server. Clearly, the ability to do this is valuable for practitioners in the field.

While our approach is based on ANUBIS, the techniques we introduced in this paper are general and can also be

realized with the help of other malware analysis platforms (such as BitBlaze [43]).

VIII. Limitations

Adversarial code is difficult to analyze. Our system needs to observe a sample's malicious activities inside our analysis environment. That is, we need to see a behavior in the ANUBIS system in order to collect the relevant log files for starting our extraction process. Thus, attacks against the dynamic analysis environment or the taint analysis are a concern for us.

ANUBIS is based on an unaccelerated version of the system emulator QEMU. While standard techniques for detecting virtual machines do not apply to this tool, it might be possible to detect the analysis environment using other means (e.g., emulator specific hardware names, ANUBIS-specific artifacts, etc.). Emulator detection versus stealthy analysis is a continuing arms race, and detectability is currently a limitation of dynamic analysis environments. As a possible solution, and to address emulator checks, we can attempt to resort to stealthy analysis techniques such as multi-path exploration [44].

As mentioned in previous sections, our system is conservative in the sense that we only include instructions we have seen during the recorded execution and fix branches in the gadget accordingly. This can cause undesired side-effects as input during gadget execution could require the original, excluded code paths. As a result, the gadget's behavior may differ from the behavior of the malware when processing that input. This could be improved by statically analyzing excluded code regions and include them if possible.

Evading taint tracking is problematic for the features of our system that rely on data tainting (e.g., the gadget inversion). However, note that the extraction of algorithms and the generation of gadgets is not dependent on data tainting. Thus, the core parts of the systems can deal with this kind of evasion attempts.

Further, our current slicing algorithm works on single threads. If multiple threads interleave execution by providing data to, or modifying data from each other, we cannot handle this situation. However, an improved tracking of threads and their interdependencies can overcome this shortcoming, and would not require conceptual improvements.

Although our prototype implementation has some limitations, our evaluation results shows that we can successfully operate on complex, real-world malware samples. Thus, we believe that our approach is useful for security practitioners in many different ways.

IX. Conclusion

Unfortunately, malicious software (i.e., malware) is still a major threat on the Internet today. In fact, malware has become the main driving force behind many attacks. Unlike a decade ago, malware-based attacks are mainly aiming to make a financial profit, and the attackers are targeting Internet users with the goal of using the victims' compromised machines for sending spam, launching denial of service attacks, and stealing confidential data.

In this paper, we improve the state of the art in malware analysis by presenting a novel approach to automatically extract, from a given malware binary, the instructions that are responsible for a certain activity of the sample. These instructions, which we call *gadgets*, encapsulate a specific behavior that can autonomously perform a particular malicious task (e.g., such as domain generation for command and control).

Our approach is valuable for analysts in the field as understanding a certain activity that is embedded in a malware sample (e.g., the update function) is still largely a manual and difficult task.

Acknowledgments

This work has been supported by the Austrian Science Foundation (FWF) and by Secure Business Austria (SBA) under grants P-18764, P-18157, and P-18368, by the European Commission through project FP7-ICT-216026-WOMBAT, by the ONR under grant no. N000140911042, and the National Science Foundation (NSF) under grant no. 0845559.

References

[1] C. Linn and S. Debray, "Obfuscation of Executable Code to Improve Resistance to Static Disassembly," in *Conference on Computer and Communications Security (CCS)*, 2003.

[2] I. Popov, S. Debray, and G. Andrews, "Binary Obfuscation Using Signals," in *USENIX Security Symposium*, 2007.

[3] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection ," in *23rd Annual Computer Security Applications Conference (ACSAC)*, 2007.

[4] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding Malware Analysis Using Conditional Code Obfuscation," in *Network and Distributed System Security Symposium (NDSS)*, 2008.

[5] L. Cavallaro, P. Saxena, and R. Sekar, "On the Limits of Information Flow Techniques for Malware Analysis and Containment," in *5th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[6] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna, "Your Botnet is My Botnet: Analysis of a Botnet Takeover," in *Conference on Computer and Communication Security (CCS)*, 2009.

[7] F. Leder and T. Werner, "Containing Conficker: Conficker Domain Name Generation," http://net.cs.uni-bonn.de/wg/cs/applications/containing-conficker, 2009.

[8] M. Ligh and G. Sinclair, "Malware RCE: Debuggers and Decryptor Development ," Defcon 16, 2008.

[9] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing," in *Conference on Programming Language Design and Implementation (PLDI)*, 1990.

[10] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary Code Extraction and Interface Identification for Security Applications," in *Network and Distributed Systems Symposium (NDSS)*, February 2010.

[11] U. Bayer, "Anubis: Analyzing Unknown Binaries," http://anubis.iseclab.org, 2009.

[12] J. Newsome and D. X. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software," in *Network and Distributed System Security Symposium (NDSS)*, 2005.

[13] X. Zhang, R. Gupta, and Y. Zhang, "Precise Dynamic Slicing Algorithms," in *International Conference on Software Engineering (ICSE)*, 2003.

[14] F. Bellard, "Qemu: A Fast and Portable Dynamic Translator," in *Usenix Annual Technical Conference, Freenix Track*, 2005.

[15] U. Bayer, C. Kruegel, and E. Kirda, "TTAnalyze: A Tool for Analyzing Malware," in *Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[16] U. Bayer, P. Milani Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering," in *Network and Distributed System Security Symposium (NDSS)*, 2009.

[17] C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and Efficient Malware Detection at the End Host," in *18th Usenix Security Symposium*, 2009.

[18] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace, "ReFormat: Automatic Reverse Engineering of Encrypted Messages," in *14th European Symposium on Research in Computer Security (ESORICS)*, 2009.

[19] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-based Fault Isolation," in *14th ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[20] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *15th USENIX Security Symposium*, 2006.

[21] W. Sun, Z. Liang, V. Venkatakrishnan, and R. Sekar, "One-way Isolation: An Effective Approach for Realizing Safe Execution Environments," in *Network and Distributed Systems Symposium (NDSS)*, 2005.

[22] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in *IEEE Symposium on Security and Privacy*, 2009.

[23] T. Holz, C. Gorecki, K. Rieck, and F. C. Freiling, "Measuring and Detecting Fast-Flux Service Networks," in *Proceedings of the 15th Annual Network & Distributed System Security Symposium (NDSS)*, 2008.

[24] N. Freed and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," http://tools.ietf.org/html/rfc2045#section-6.8, 1996.

[25] P. Porras, H. Saïdi, and V. Yegneswaran, "A Foray into Conficker's Logic and Rendezvous Points," in *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[26] A. Decker, D. Sancho, L. Kharouni, M. Goncharov, and R. McArdle, "Pushdo/Cutwail Botnet: A study of the Pushdo/Cutwail Botnet," TrendMicro Labs, 2009.

[27] A. Sotirov, "Tiny PE: Creating the smallest possible PE executable," http://www.phreedom.org/solar/code/tinype/, 2006.

[28] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage, "On the Spam Campaign Trail," in *1st Usenix Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

[29] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov, "Spamming Botnets: Signatures and Characteristics," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, 2008.

[30] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy, "Studying Spamming Botnets Using Botlab," in *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[31] T. Holz, M. Engelberth, and F. Freiling, "Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones," in *European Symposium on Research in Computer Security (ESORICS)*, 2009.

[32] Finjan Malicious Code Research, "Malware Analysis - Trojan Banker URLZone/Bebloh," http://www.finjan.com/MCRCblog.aspx?EntryId=2345, 2009.

[33] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *ACM Conference on Computer and Communication Security (CCS)*, 2007.

[34] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, "Dynamic Spyware Analysis," in *Usenix Annual Technical Conference*, 2007.

[35] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. Kemmerer, "Behavior-based Spyware Detection," in *15th Usenix Security Symposium*, 2006.

[36] A. Lanzi, M. I. Sharif, and W. Lee, "K-Tracer: A System for Extracting Kernel Malware Behavior," in *Network and Distributed System Security Symposium (NDSS)*, 2009.

[37] M. Weiser, "Program Slicing," in *International Conference on Software Engineering (ICSE)*, 1981.

[38] Z. Lin, X. Zhang, and D. Xu, "Reuse-Oriented Camouflaging Trojan: Vulnerability Detection and Attack Construction," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS 2010)*, June 2010.

[39] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[40] R. Hund, T. Holz, and F. Freiling, "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms," in *18th USENIX Security Symposium*, 2009.

[41] J. Newsome, D. Brumley, J. Franklin, and D. Song, "Replayer: Automatic Protocol Replay by Binary Analysis," in *13th ACM Conference on Computer and Communications Security (CCS)*, 2006.

[42] J. Caballero, P. Poosankam, C. Kreibich, and D. Song, "Dispatcher: Enabling Active Botnet Infiltration using Automatic Protocol Reverse-Engineering," in *ACM Conference on Computer and Communication Security (CCS)*, 2009.

[43] 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 *4th International Conference on Information Systems Security (ICISS)*, 2008.

[44] A. Moser, C. Kruegel, and E. Kirda, "Exploring Multiple Execution Paths for Malware Analysis," in *IEEE Symposium on Security and Privacy*, 2007.