Towards Reducing the Attack Surface of Software Backdoors

Felix Schuster and Thorsten Holz
Horst Görtz Institute for IT-Security
Ruhr-University Bochum, Germany
{firstname.lastname}@rub.de

ABSTRACT

Backdoors in software systems probably exist since the very first access control mechanisms were implemented and they are a well-known security problem. Despite a wave of public discoveries of such backdoors over the last few years, this threat has only rarely been tackled so far.

In this paper, we present an approach to reduce the attack surface for this kind of attacks and we strive for an automated identification and elimination of backdoors in binary applications. We limit our focus on the examination of server applications within a client-server model. At the core, we apply variations of the *delta debugging* technique and introduce several novel heuristics for the identification of those regions in binary application that backdoors are typically installed in (i.e., authentication and command processing functions). We demonstrate the practical feasibility of our approach on several real-world backdoors found in modified versions of the popular software tools *ProFTPD* and *OpenSSH*. Furthermore, we evaluate our implementation not only on common instruction set architectures such as x86/x64, but also on commercial off-the-shelf embedded devices powered by a MIPS32 processor.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

General Terms

Security

Keywords

Software Backdoors; Dynamic Analysis; Binary Analysis

1. INTRODUCTION

In today's computing environment, we typically rely on software components that are not implemented by ourselves, but from a (potentially untrusted) third party, e.g., a commercial provider or an open source project. This is a common practice and natural consequence of the complexity of modern computer systems. Of course, executing unknown code has several security implications; among others it involves the question whether the executable has some undocumented, unwanted, or hidden functionality [35]. Purposefully (or even unintentionally) the program could for example have an undocumented account that is only available upon specific conditions [2], or offer another kind of remote access to an adversary [3–5]. In the best case, such code does not cause any harm at all, but it is a challenging problem to determine if an unknown piece of (binary) code contains a backdoor. From an attacker's perspective, especially server applications are an interesting target for the deployment of backdoors: such applications are typically remotely reachable and an attacker can abuse them to obtain access to a system that can afterwards be used as a stepping stone to attack further machines within a network. Hence, our focus within this paper lies on network services since we view them as some of the most relevant attack targets.

Towards Prevention of (Software) Backdoors.

While lots of work on implementing and detecting backdoors in hardware has been published recently (e.g., [6, 21, 24, 36]), the area of software backdoors has not received much attention. Previous work in that area has mainly addressed authentication routines. For example, Dai et al. introduced an approach to eliminate backdoors from *response-computable authentication* routines, i.e., the typical challenge-response based authentication [12]. Their idea is to reduce the attack surface by eliminating potential triggers for a backdoor. However, an attacker could still add a backdoor by adding additional command handlers or installing mechanisms that entirely bypass the authentication process in a server application on certain triggers.

To address the problem, the idea of *privilege separation* can be used to split an application into different trust domains (e.g., [9, 30]). Nevertheless, an attacker with broad access to an application's source code can still install backdoors within sufficiently privileged components and thus a certain attack surface remains. The problem is further complicated in cases where we do not have access to source code, but need to perform our analysis on the binary level only.

Automated Identification of Sensitive Components.

In this paper, we address the problem of software backdoors in server applications and introduce an automated way

CCS'13, November 4–8, 2013, Berlin, Germany.

ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516716.

to detect and to disable certain types of backdoors in a given binary program. Our approach consists of three phases:

In the first phase, we identify the specific regions in a given binary that are especially prone to attacks, i.e., the authentication routines and code related to command dispatching and command handling functionality. By closely monitoring the server application while it processes inputs automatically generated according to a protocol specification and analyzing the resulting traces, we can spot these parts in a precise and automated way. We leverage the idea of *delta debugging/differential analysis* [16,40] and introduce an algorithm to identify the relevant regions of a given binary application.

In the second phase, once the initial components of interest are identified, we use this knowledge to determine suspicious components in an application, e.g., hidden command handlers or edges not taken in the control flow graph (CFG) of an authentication routine. To this end, we introduce several heuristics that enable us to determine which code regions are suspicious. Furthermore, we aggregate information that can serve as a starting point for further automated (or manual) investigations.

In an optional third phase, the results from the previous phases can be used to modify or instrument the server application in an automated way such that program parts identified as suspicious are monitored or disabled at runtime. As a side effect, this feature can be used as a flexible way to enable dynamic reconfiguration of server applications for which no source code is available.

Results.

We have implemented our approach to detect and to disable software backdoors in server applications in a tool called WEASEL. For the recording of traces and other runtime analyses, WEASEL employs `gdb`, the standard debugger for the GNU software system. `gdb` is available for a large number of operating systems and hardware architectures and WEASEL currently contains adapter code for x86, x64 and MIPS-32 processor architectures running Linux operating systems, enabling the analysis of a wide range of platforms.

We have successfully tested WEASEL with seven different server applications on different platforms including a widespread corporate VoIP telephone and a popular consumer router. In all cases, we were able to precisely and automatically identify the key program parts involved in the authentication process or the dispatching and handling of commands. We were able to detect known real-world backdoors contained in certain versions of ProFTPD and OpenSSH [2, 15]. Furthermore, as a case study, graduate students from our group have implemented eleven different kinds of backdoors for ProFTPD. Our tool can be used to detect or disable the majority of these artificial backdoors as well. This demonstrates the practical feasibility of the approach to reduce the attack surface for software backdoors.

To demonstrate a potential mitigation approach, WEASEL is also capable of transforming a given binary application to reduce the set of available commands. This is implemented by precisely identifying the command dispatcher functions leveraging automated data structure identification methods [25,31]. Once we have found the specific data structures, we modify them such that certain commands are inaccessible.

In summary, this paper makes the following contributions:

- We introduce an automated way to identify critical parts in server applications that are typically prone to backdoors. We show how this information can be used a) to detect backdoors and hidden functionality and b) to reduce the attack surface for backdoors in legacy binary software.
- We implemented our techniques in a tool called WEASEL for x86, x64 and MIPS32 systems running different versions of Linux. The code is freely available at `https://github.com/flxflx/weasel`.
- We show how several real-world backdoors for ProFTPD and OpenSSH can be successfully discovered using our techniques. Furthermore, we evaluated our techniques with other examples of binary programs on different platforms, including COTS embedded devices.

2. RELATED WORK

Backdoors in computer systems are a well-known security problem. Over the last few years, the security community has mainly focused on backdoors in integrated circuits and ways to detect or implement such hardware backdoors [6,19, 21,24,33,34,36]. Furthermore, several backdoors in different components of a computer such as network cards [32] or directly in the CPU [14] were proposed. Our approach is orthogonal to such work since we focus on the detection of backdoors on the binary level, an area that has received almost no attention so far.

Wysopal et al. [37] presented a pattern-based, static analysis approach to identify backdoors in software. The main limitation is that patterns need to be specified in advance, which implies that potential backdoors need to be known before the analysis can be carried out. Nevertheless, the results delivered by our tool could be used to improve the accuracy of such a static analysis approach: for example, a static ASCII string should be more likely to represent a static password when it is referenced in or close to a function identified to be involved in the authentication process of an application.

Geneiatakis et al. [18] recently proposed a similar (although not backdoor-related) technique to ours to identify *authentication-points* in server applications using Intel's *Pin* tool [26]. Our approach can be applied to a broader range of platforms and environments and is fully automated. In fact, our implementation only requires a remote `gdb` instance to connect to and automatically collects all required traces on function and basic block level, while invoking the corresponding server application repeatedly without requiring any supervision. By only relying on `gdb`, we are able to analyze software on COTS embedded devices such as routers or VoIP telephones without special effort. We think that this is especially useful in the context of backdoors given the often high criticality of such devices and the rather large amount of corresponding backdoors publicly reported lately [3–5]. Furthermore, we consider *implicit edges* in the CFG of a function induced by conditional instructions as described in Section 4, a problem not addressed in prior work.

A basic insight for backdoor detection is that some kind of trigger needs to be present such that an attacker can activate the backdoor. As a result, work on automated identification or silencing of triggers is also related to our work. Brumley et al. [10] demonstrated how trigger-based behavior in malware can be identified and Dai et al. introduced an approach to eliminate backdoors from *response-computable authentication* routines, i.e., the typical challenge-response based authentication [12]. While such approaches reduce the attack surface, an attacker can still implement a backdoor and bypass the approach (e.g., by adding additional command handlers or completely bypassing the authentication process). Our approach complements such approaches and helps to reduce the attack surface even further.

Somewhat related to our approach is the idea of *privilege separation*, i.e., the process of splitting an application into different trust domains [9, 11, 22, 28, 30, 39]. Note that such approaches do not completely mitigate the risk of backdoors since an attacker can still install backdoors within sufficiently privileged components. The backdoors for SSH servers analyzed in Section 5 demonstrate that this is indeed a problem in practice.

3. APPROACH

We now provide a high-level overview of our approach before presenting implementation details in the next section.

3.1 Attacker Model for Software Backdoors

There are many different mechanisms an attacker can use to bypass implemented security mechanisms. To narrow down the scope of the discussions, we first define the term *backdoor* in the context of this paper to illustrate the capabilities of an attacker:

Definition: *A backdoor is a hidden, undocumented, and unwanted program or a program modification/manipulation that on certain triggers bypasses security mechanisms or performs unwanted/undocumented malicious actions.*

The definition should be seen as an intuitive and tangible description of what a backdoor is. Note that a backdoor can be either *intentional* or *unintentional*, depending on whether it was created on purpose or as a dangerous side-effect of some other "harmless" activity (e.g., debugging access with elevated privileges that is not disabled in the release version of a product). Naturally, there are unlimited ways how an attacker can implement a backdoor in a given software system: an adversary can for example purposely introduce buffer overflow vulnerabilities or flawed cryptographic modules, both of which complicate the analysis significantly. In general, the problem of detecting software backdoors is undecidable and thus we strive for an automated way to at least reduce the attack surface and detect certain cases of backdoors in practice.

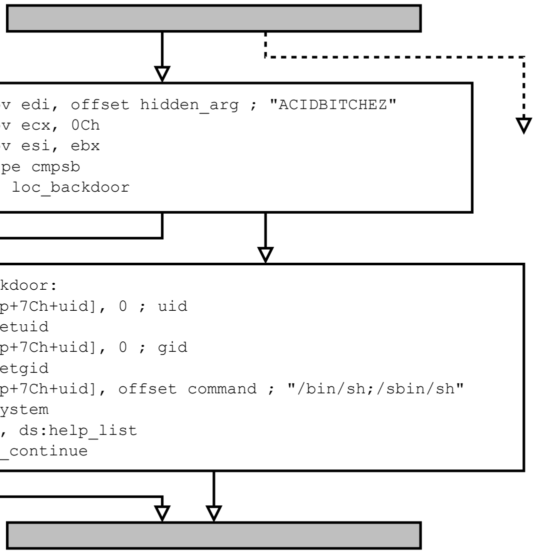
For the remainder of this paper, we focus on the following classes of backdoors as part of our attacker model:

- flawed authentication routines
- hidden commands and services

In fact, most of the recently publicly disclosed backdoors fall into these two classes [2–5, 15, 20, 27] and thus we think that this constraint for the problem space is reasonable.

Running Example.

As our running example, we illustrate a backdoor that attackers added to the ProFTPD server. This example highlights the challenges we face and explains some of the issues we have to deal with. Note that the example is in C code, while we perform our analysis on the binary level.



```
mov edi, offset hidden_arg ; "ACIDBITCHEZ"
mov ecx, 0Ch
mov esi, ebx
repe cmpsb
jz loc_backdoor
```

```
loc_backdoor:
mov [esp+7Ch+uid], 0 ; uid
call _setuid
mov [esp+7Ch+uid], 0 ; gid
call _setgid
mov [esp+7Ch+uid], offset command ; "/bin/sh;/sbin/sh"
call _system
mov eax, ds:help_list
jmp loc_continue
```

Figure 1: The two additional basic blocks in the flow graph of the function *pr_help_add_response()* in ProFTPD implementing the `ACIDBITCHEZ` backdoor

At the end of November 2010, the distribution server of the ProFTPD project was compromised and a snippet of code was added to one of the source files of ProFTPD 1.3.3c [2]. In essence, a function responsible for the processing of the standard FTP command HELP was modified in such a way that passing the argument `ACIDBITCHEZ` would result in immediate privileged access to the corresponding system (a *root shell*) for an unauthenticated user. The actual malicious code introduced by the attacker is shown in Listing 1. The backdoor was (likely manually) detected about three days later and then eliminated by removing the changes.

Listing 1: Backdoor in ProFTPD server

```
if (strcmp(target,"ACIDBITCHEZ") == 0)
{
  setuid(0);
  setgid(0);
  system("/bin/sh;/sbin/sh");
}
```

The central goal of our approach is to automatically identify such harmful extensions. The scope of our approach is limited to binary code, mainly since we often do not have access to source code.

Malicious additions to binary software can often be reliably detected by static analysis techniques [16] when at least one supposedly benign version of the software is available. Typically the deployment of a simple backdoor introduces a handful of additional basic blocks to the CFG of a function that handles external inputs. Figure 1 depicts the malicious addition to the CFG of the function `pr_help_add_response()` in ProFTPD that constitutes the backdoor of our running example. In such a case, binary software comparison approaches [16, 17] are likely capable to detect these suspicious code regions. Unfortunately, in practice we typically do not have access to such a version that can reliably be trusted for most software systems. This is why our approach does not rely on the existence of such a version. Instead, we examine the behavior of a single version of binary software at runtime and apply techniques that extend the idea of Zeller's *delta debugging* approach [40] on binary level as explained in the following.

3.2 Analysis Approach

We argue that there are four parts of a server application that are generically prone to the two backdoor classes in our focus. These parts are:

- authentication validation code
- specific authentication validation result handling code (e.g., code that terminates a session in case of invalid credentials)
- command parsing code
- specific command handling code

To identify these parts in a given server application, we record runtime traces for multiple different inputs and compare them. The intuition behind this approach is the following: consider for example the authentication mechanism of a server application such as ProFTPD. By definition, the purpose of each authentication mechanism is to decide whether or not a third party sufficiently proved its identity to qualify for legitimate elevation of privilege. The process is similar in case of command dispatching: different operations are performed for different commands and arguments. In order to behave differently for different inputs, a server application in general needs at one point during runtime to leave the common execution path and follow an exclusive execution path accordingly[1]. By comparing control flow traces for various inputs, it is possible to determine *common* execution paths and those that are *exclusive* to a certain group of inputs. In the next step, it is often possible to determine *deciders* and *handlers* on function or on basic block level. In the case of the authentication process, *deciders* perform the actual authentication validation, while *handlers* process the validation result. In the case of the command dispatching process, *deciders* parse and dispatch commands, while *handlers* implement the commands' specific functionality.

In general, the following possibilities exist for a server application to implement exclusive execution paths for different inputs:

C1 through exclusive function invocations

C2 through exclusive paths inside commonly invoked functions (i.e., exclusive basic blocks)

C3 without exclusive program parts, but through an exclusive execution order of common functions and basic blocks

In our empirical studies of different server applications we found that the two cases C1 and C2 are by far the most common in practice. Case C3 is not entirely unlikely to be encountered, though: consider for example a server application implementing the available commands in an internal scripting language. Runtime traces both on function and on basic block level for different inputs of such a server application would differ, but possibly not contain any exclusive program parts.

Cases C1 and C2 will receive the most attention for the rest of this work. An intuitive and straightforward approach for the identification of *handlers* and *deciders* in these cases is the following: Given two traces T_0 and T_1 for different inputs (e.g., valid password and invalid password), *handlers* can be identified by determining the set of exclusive functions/basic blocks for each of the two traces: $S_{T_0,ex} = S_{T_0} \backslash S_{T_1}$ and $S_{T_1,ex} = S_{T_1} \backslash S_{T_0}$. In turn, the corresponding *deciders* are necessarily in $S_{common} = S_{T_0} \cap S_{T_1}$ and are likely parents of exclusive functions/basic blocks in $S_{ex} = S_{T_0,ex} \cup S_{T_1,ex}$.

Built upon this basic idea for the identification of *deciders* and *handlers*, we have developed the algorithm A-WEASEL which is described in the following.

3.2.1 The A-WEASEL Algorithm

The A-WEASEL algorithm is an integral part of our approach. It is described in detail in Appendix A and we provide a high-level overview now. The algorithm starts working on function level traces only since they can be collected on virtually any platform where `gdb` (but possibly no instrumentation solution like Pin) is available in a straightforward and efficient way. Basic block level traces are collected as needed.

Given a set of traces of a server application on function level for different *protocol runs* (e.g., traces for different FTP commands), we recursively compute a combined *decision tree* composed of *deciders* and top-level *handlers* as depicted in Figure 2. Handlers are initially identified by simply determining the set $S_{T,ex} = S_T \backslash S_{common}$ of exclusive functions for each trace and can be shared between multiple traces. For each identified decider (or handler shared between multiple traces) on function level, traces on basic block level are recorded dynamically for all corresponding protocol runs[2]. Given these basic block traces, we compute the *internal decision tree* of the respective function. For example in Figure 2 the internal decision tree of the decider function `z` is generated from corresponding basic block level traces for protocol runs 1 and 2. A-WEASEL is recursively invoked with the applying set of function level sub-traces for each identified decider function.

A-WEASEL reliably determines the decision tree for a given set of traces, though it is only applicable for the aforementioned server application implementation case C1 or a combination of cases C1 and C2. This is due to the algorithm depending on the identification of exclusive handlers: if there are no functions that are exclusive to a certain subset of all traces, no handlers can be identified (and as a consequence, no deciders as well). In order to cope with server applications that are implemented strictly according to cases C2 and C3 as well and to improve overall results, we implemented a set of additional algorithms in WEASEL that are outlined in the following.

[1] One could probably draw scenarios where the different features of a server application are entirely implemented by differences in the data flow only and not in the control flow. Due to the lack of real-world relevance of such scenarios, they are not considered further here.

[2] We record basic block level traces of functions in a call stack sensitive manner. This way we can ensure that only the certain invocations of interest of a function are examined on basic block level.

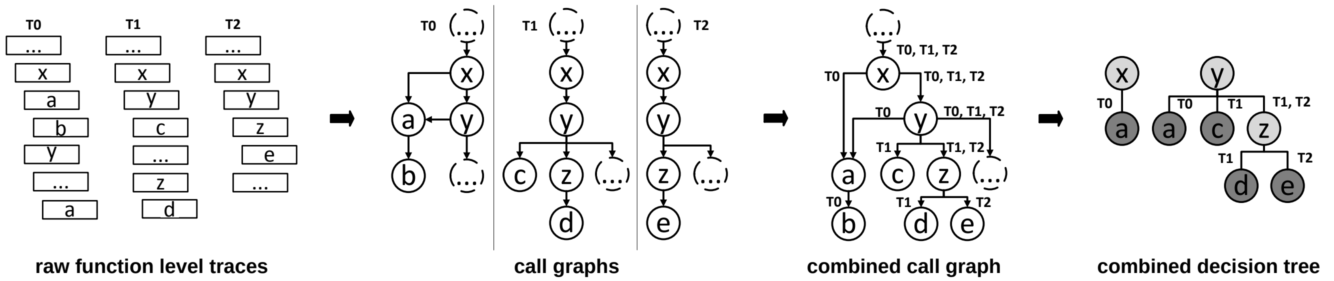


Figure 2: Schematic derivation of the *decision tree* (right) from the examplary function level traces $T1$, $T2$ and $T3$ (left). The functions a, b, c, d, e and z are not contained in all traces and are thus *exclusive* to certain traces. The functions x, y and z are *deciders*, while a, c, d and e are top-level *handlers*.

Differential Return Value Comparison.

A major drawback of A-WEASEL is that it fails to identify *decider* and *handler* functions when decisions made at runtime only manifest in differences in the execution flow on basic block level. To cope with this problem, WEASEL records exactly two traces for each protocol run. Functions with different return values in both traces are added to the set of functions to ignore. This way, irrelevant functions in our context such as `malloc()` or `time()` are filtered out. Functions with the same return value in both traces, but different return values between different protocol runs, are treated as handlers by A-WEASEL. By applying this technique, A-WEASEL for example correctly identifies the function `sys_auth_passwd()` as decider in the authentication process of OpenSSH (see Section 5.1.1), which it would have not otherwise. Though there are numerous ways a function can signal its outcome to its caller, the described technique only takes immediate return values of functions into account.

Scoring Heuristics.

WEASEL contains a set of simple heuristic scoring algorithms that aim at identifying *deciders* and estimating their importance by comparing the structure of the call graphs of a given set of traces. The algorithms are used to rank the importance of *deciders* identified by A-WEASEL. Besides, they can partly serve as a fall back when server applications are encountered that do not implement exclusive functionality through exclusive function invocations (cases C2 and C3 as discussed above). All algorithms have in common that they require a *reference trace* which the other traces are compared to. When examining the authentication process, the trace for valid credentials serves as reference trace, while in the case of the command dispatching process a specially recorded trace for a knowingly invalid command serves this purpose.

For instance, one algorithm attempts to remove loops from all traces, determines the longest common subsequence of calls between all traces and the reference trace and assigns scores to functions according to their positions in the common sub-sequences (scores increase towards the end as those functions are believed to be more likely to be responsible for the decision to split execution paths). An even simpler algorithm assigns scores to deciders linear to the amount of exclusive children. The scores assigned to functions by the different algorithms are summed up.

3.3 Application of Analysis Results

Once the *deciders* as well as the *handlers* are known for the authentication or the command dispatching process of a server application, further analysis can be conducted. The goal is to identify possible backdoors and to enable the hardening of legacy binary applications.

3.3.1 Discovering Suspicious Program Paths

When the functions or basic blocks handling a successful authentication or a certain command are known, we can apply existing methods of static and dynamic binary analysis for the detection of backdoors. A straightforward approach which we apply here is the static enumeration and comparison of all syscalls and external library calls reachable in the static call graph from identified handlers (we utilize the third party tool *IDA Pro* for this). For example, even invocations of `socket()` or `send()` should be considered suspicious when they are only referenced from one of multiple handlers. In the case of our running example, the installed backdoor in the HELP command can be identified this way (see Section 5.2 for a detailed discussion). Moreover, identified deciders and handlers can be used as starting points for more complicated analysis techniques such as *symbolic execution* [23]. Starting symbolic execution at the entry-point of identified handler code should in many cases – and especially for complex software such as ProFTPD – deliver better and more cost-effective results than approaches examining an entire application. In order to be able to apply techniques of symbolic execution, one of course always needs to declare certain memory as symbolic. Identifying memory regions that are worthwhile to declare as symbolic poses a challenge when examining single functions. To tackle this problem, we have implemented an analysis module for WEASEL that compares the arguments of identified deciders in different traces and heuristically searches for differences (see Section 4.2.2). In the case of a typical password validation function it is then for example possible to determine that certain arguments are pointers to memory regions with varying contents (e.g., username and password). In the next step, these memory regions could be marked as symbolic when analyzing the respective function with symbolic execution techniques.

3.3.2 Disabling Functionality

One can very well think of scenarios where it is desirable to disable certain functionality of a legacy server application. For example it might be known that certain commands are vulnerable to attacks. Instead of shutting the whole service down or applying error-prone filtering on the network level, our approach allows for the disabling of single commands by means of binary instrumentation or binary rewriting. In the case of our running example, effective protection can already be achieved by simply writing an illegal instruction at the start of the handler for the HELP command, causing the respective fork of the server to crash and exit when the command is issued.

Backdoors in the authentication process, like for example hardcoded credentials, often manifest in additional edges and nodes in the CFG of one of the involved decider functions. These additional edges and nodes are usually not contained in any recorded basic block trace for legitimate input. We call such edges "cold". For complex software, cold edges and nodes are only a rather weak indication for the presence of a backdoor, as there are usually many benign basic blocks that are only visited under rare conditions. Nevertheless, knowledge of cold edges in decider functions of the authentication process can be used to increase the protection level of applications: techniques for binary instrumentation or rewriting can for example be used to log access to edges identified as *cold* during runtime over a longer period. In case an edge is taken for the first time, an alert can be triggered and the incident can be investigated. In practice, we suggest the utilization of a training phase during which additional benign paths are discovered and successively enabled before the final rewriting/instrumenting takes place. Entirely disabling cold edges in decider functions might severely weaken the security of an application, e.g., the protection against password brute-forcing could be rendered non-functional. In the following, we use the term of "cutting an edge" in order to refer to the monitoring or disabling of an edge.

Another application of our approach is the identification and elimination of undocumented commands. Command deciders of server applications of classes C1 or C2 dispatch recognized commands either through conditional and static (e.g., JZ OFFSET) or dynamic (e.g., CALL EAX) branches to their designated handlers. The latter is the case for our running example: when a command is recognized in ProFTPD, a *C* structure describing it is loaded from a static table. Each such structure contains a pointer to the handler function for the corresponding command, which is called by the decider/dispatcher through a CALL EAX instruction. For server applications built in a similar way, two interesting measures become possible on top of our basic approach:

- Once several command handlers are known, a likely location and size of the table(s) holding the command-describing structures in memory can be determined. In the next step, it is possible to identify all available commands and unwanted commands can easily be eliminated using techniques of binary instrumentation or rewriting.

- Once the point of execution is known where the control flow is dynamically transferred to a command handler, techniques of binary instrumentation or rewriting can be used to prevent the execution of unknown or unwanted command handlers.

We have developed a module for our analysis framework that heuristically checks for tables containing command descriptors given a set of pointers to command handlers (see Section 4.2).

3.3.3 Enforcing Authentication

When deciders and handlers of the authentication process of an application can be linked to certain authentication levels, it becomes possible to determine the authentication level of an active session by examining the execution flow of the corresponding thread at runtime. Combined with the knowledge of whereabouts of command handlers, fine-grained access control or defense mechanisms such as *shadow authentication* [13] can be realized. In our running example, it would be possible to limit the availability of the HELP command to those threads that were observed successfully authenticating before.

4. IMPLEMENTATION

The core of the analysis framework WEASEL is our library PYGDB. It is written for Python 2.7 and implements a client for the *GDB Remote Serial Protocol* [1] and thus needs to be connected to a remote `gdbserver` instance. PYGDB supports all basic debugging tasks from setting breakpoints to following forks. Currently our software supports environments running Linux on x86, x64, or MIPS32 platforms.

The tracing engine is built on top of PYGDB and supports tracing on function as well as on basic block level. As PYGDB is designed to not include comprehensive disassemblers for its supported platforms, basic blocks are initially identified by stepping single instructions.

The tracer is aware of *implicit edges* in the CFG induced by conditional instructions and records *virtual basic blocks* accordingly. To understand the need for this consider the x86 conditional move instruction `CMOVZ`: data is only moved from the source to the destination operand in case the *zero-flag* is set (e.g., as the result of a compare operation). Compilers use such instructions to efficiently translate simple *if-then* constructs. Taking this into account can be crucial for successfully detecting and disabling backdoors (see Sections 5.1.1 and 5.2).

4.1 Protocol Player

In order to fully automate the analysis process, we developed a system for the specification and playback of protocols. Similar to existing work in the realm of fuzz testing of software [7,8], we describe protocols in a block-based manner according to their specifications. The blocks describing a protocol are ordered in *levels* and are grouped by *strings* and *privilege levels*. Our description of the FTP protocol according to RFC 959 [29] for example possesses nine levels, the strings CMDARG0, CMDARG1 and CMDARG2 and the privilege levels NOAUTH, ANONYMOUS and AUTH.

Before traces of a certain server application are recorded, the corresponding *protocol description* is compiled to a set of specific *protocol scripts*. Compiled scripts are solely built of the atoms PUSH_DATA, SEND, RECV and WAIT, and the virtual atoms START_RECORDING and STOP_RECORDING. The last two are automatically inserted by the *protocol compiler* before and after the atoms of interest. When the *protocol processor* encounters one of them while playing the protocol, it activates/deactivates the tracer. We thus ensure that as little noise as possible is recorded. In order to be able to describe interactive elements in protocols (such as for example the PING/PONG messages in the IRC protocol), each atom can dynamically yield new atoms in reaction to the state of the protocol run.

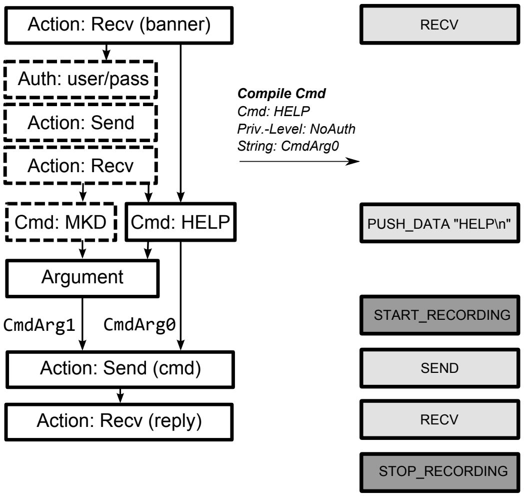


Figure 3: Scheme of the description of the FTP protocol (two commands) and the correspond compiled script of the command HELP. Blocks not available for privilege level NoAuth are dashed. Atoms are gray. Virtual atoms are dark-gray.

Figure 3 schematically shows the description of two commands of the FTP protocol: the command HELP accepts none or one argument. Accordingly, it belongs to the strings CmdArg0 and CmdArg1. As the command is available in any session – unauthenticated as well as authenticated – it belongs to all three privilege levels. In contrast, the command MKD belongs to the privilege levels Anonymous and Authenticated, and solely the string CmdArg1 as it expects one argument and is only available in authenticated sessions.

4.2 Analysis Modules

We implemented two analysis modules that work on the results delivered by the A-WEASEL analysis algorithms described in Section 3.

4.2.1 Function Pointer Table Identifier

Many server applications written in C/C++ store command descriptors including pointers to *handler* functions in central data structures such as arrays. We have implemented an analysis module that scans the memory of a server application at runtime for pointers to previously identified handler functions. When the distance in memory between several identified pointers to handler functions is of equal size, we assume that a table of command descriptors was found (compare [25, 31]). In the next step, we attempt to heuristically determine the beginning and end of the respective table. Once such a table is identified it is possible to check for pointers to unknown command handlers and thus identify undocumented commands.

4.2.2 Differential Function Arguments Identifier

There are several scenarios in which it might be desirable to identify those arguments of a *decider* function that are protocol run specific (see Section 3.3.1). A simple example is a password validation function that expects (among other not session specific arguments) pointers to both the username and the password entered by a user. We implemented an analysis module that – given the list of decider functions of a server application – tries to heuristically identify such arguments. The module replays the different protocol runs and examines the stack at the entrance to the given decider functions in a differential manner. The module thus only works for calling conventions passing arguments on the stack. The module distinguishes between *data* and *pointers* and tries to identify pointers by dereferencing values in memory and checks if the resulting address resides in the same type of memory for each protocol run. Thereby, the module differentiates between the following types of memory: stack, heap, and binary image. If a value in memory is found to be pointing to the same type of accessible memory for each protocol run, it is assumed that it is in fact a pointer. Otherwise it is assumed to be plain data. In this case, the data is compared between all protocol runs. In case of any difference, the argument is marked as protocol run specific. The analysis module follows alleged pointers up to a certain level of depth. This way it is possible to identify protocol run specific arguments passed inside of nested data structures.

5. EVALUATION

To demonstrate the practical feasibility of our approach, we evaluated our analysis framework Weasel with several open and closed source server applications for different protocols and platforms. The results are briefly summarized in Table 1. All applications were tested in a standard configuration. For the sake of simplicity, Weasel was limited to only consider traces of login attempts for the following cases: Valid username/valid password (*valid-pw*) and valid username/invalid password (*invalid-pw*). In its default configuration, Weasel also considers the third case of an invalid username. This can for example be useful for the detection of backdoors triggering on certain usernames. The amount of function call events in a single trace ranged from 3 (BusyBox Telnetd authentication) to 12,335 (ProFTPD command dispatching). Due to space constraints, we discuss the test results of only three server applications in more detail in the following. For the other four applications, the test results were satisfactory to the extent that Weasel correctly identified handlers and deciders of both the authentication and the command dispatching process with little to no noise.

5.1 Detailed Analysis of SSH Servers

Due to their practical relevance, we first focus on software backdoors for SSH server implementations. The description of our SSH protocol is limited to the *SSH Authentication Protocol* (SSH-AUTH) according to RFC 4252 [38]. Other aspects of the SSH protocol, such as the transport layer, are not considered for our purposes. RFC 4252 specifies the following authentication methods for SSH-AUTH: *password*, *publickey*, *hostbased*, and *none*. The corresponding protocol specification in Weasel treats these four methods as commands.

5.1.1 OpenSSH (x64)

For OpenSSH we chose a version that was recently reported by an antivirus vendor to have been found in the wild containing the following backdoors [15]:

Table 1: Overview of evaluation results for various server applications. The *decision tree* column describes the calculated command dispatching and authentication decision trees in the form *<number of decider functions>/<number of handler functions>*.

Server	Platform	Protocol	Dec. Tree (Cmd, Auth)	Remarks
BusyBox Telnetd	MIPS32 (D-Link DIR-300)	Telnet	1/0, 0/0	Does not support Telnet commands (IAC).
Dropbear SSH	MIPS32 (Siemens OpenStage 40)	SSH-AUTH	3/9, 1/2	See Section 5.1.2
OpenSSH	x64	SSH-AUTH	2/2, 4/7 (monitor proc.) 2/3, 3/3 (slave proc.)	See Section 5.1.1
ProFTPD	MIPS32, x86, x64	FTP	6/60, 5/37 (x64)	See Section 5.2
Pure-FTPd	x64	FTP	2/29, 1/11	Results similar to unmodified ProFTPD
NcFTPD (closed source)	x86, x64	FTP	4/40, 2/11 (x64)	Results similar to unmodified ProFTPD
Dancer-IRCD	x64	IRC	2/28, 2/12	Command handler table automatically identified.

X1 On startup, the server sends the hostname and port on which it is listening to remote web hosts presumingly controlled by the attackers.

X2 A master password enables logins under arbitrary accounts without knowledge of the actual corresponding passwords (*password* and *keyboard-interactive* authentication).

X3 A master public key enables logins under arbitrary accounts with knowledge of the corresponding private key (*publickey* authentication).

X4 Credentials used in any successful login attempts are sent to the remote web hosts (*password* and *keyboard-interactive* authentication).

As no source code is publicly available for this malicious version of OpenSSH, we had to limit our evaluation to the x64 platform. Due to the privilege separating architecture of OpenSSH [30], Weasel automatically generates decision trees for two processes: a privileged process called *monitor* and an unprivileged process called *slave*.

Authentication.

Backdoor X4 can be easily spotted from the decision tree of the monitor process for the SSH *password* authentication as depicted in Figure 4. The decision tree only contains the decider functions located at virtual addresses `40B440h` (`auth_password()`), `420E20h` (`mm_answer_authpassword()`), `412EB0h` (`auth_log()`), and `40B390h` (`sys_auth_passwd()`) in the binary file[3]. The scoring algorithms of Weasel rank the decider `auth_password()` as most important. It leads to five exclusive handlers for *valid-pw* that are all called from the same handler basic block and implement backdoor X4. Of these exclusive handlers, the one at addresses `43BF50h` can automatically be identified as highly suspicious, as it (among others) statically calls the functions `socket()`, `connect()` and `write()`. Manual analysis reveals that this handler function attempts to send data to remote web hosts. Correspondingly, the handler function at address `43BAF0h` implements URL encoding of strings.

The basic block level decision tree of `auth_password()` contains 13 deciders and two exclusive handlers for the *valid-pw* protocol run (see Figure 4). While one of the handlers contains backdoor X4 as described above, the other handler is a legitimate virtual basic block induced by the conditional assembly instruction `SETNZ DL`, which sets the return value of the function according to the validity of the password. Of the 13 deciders in the basic block level decision tree, eleven are cold. Most importantly, these cold edges are related to the optional PAM authentication[4], password expiration handling and backdoor X2 (*master password*). The attacker implemented this backdoor by adding a short piece of code at the beginning of `auth_password()`: each password to check is compared to a predefined one. In case of a match, the function returns, falsely indicating a successful authentication to its caller. The backdoor is automatically rendered inoperative by *cutting* (see Section 3.3.2) the cold edges in `auth_password()`. As Weasel's protocol description of SSH-AUTH does not cover the *publickey* authentication method, we cannot find the backdoor X3. Note that this is only a limitation of the current protocol description as the implementations of backdoor X3 and backdoor X2 are very similar on assembler level. Weasel cannot be used to identify backdoor X1 (*notification of remote web hosts on startup*), because it is designed to only examine the authentication and command dispatching processes of server applications.

The decision tree of the slave process is not depicted. It consists of three deciders, two handlers for *invalid-pw* and a single handler for *valid-pw*. As static analysis hints at nothing suspicious in these functions, the slave process is not further discussed here.

Command Dispatching.

As for the authentication, command dispatching in OpenSSH stretches over a monitor and a slave process. For the slave process, Weasel identifies only the function at `414960h` (`input_userauth_request()`) as decider. Also, the deci-

[3] The actual malicious binary file does not contain debugging symbols and thus names of function cannot be obtained directly. For reasons of clearness, the names of functions of interest for this paper were manually resolved by comparing assembly code and OpenSSH source code.

[4] Code path related to PAM were not taken during testing, as PAM was not enabled in our employed default configuration.

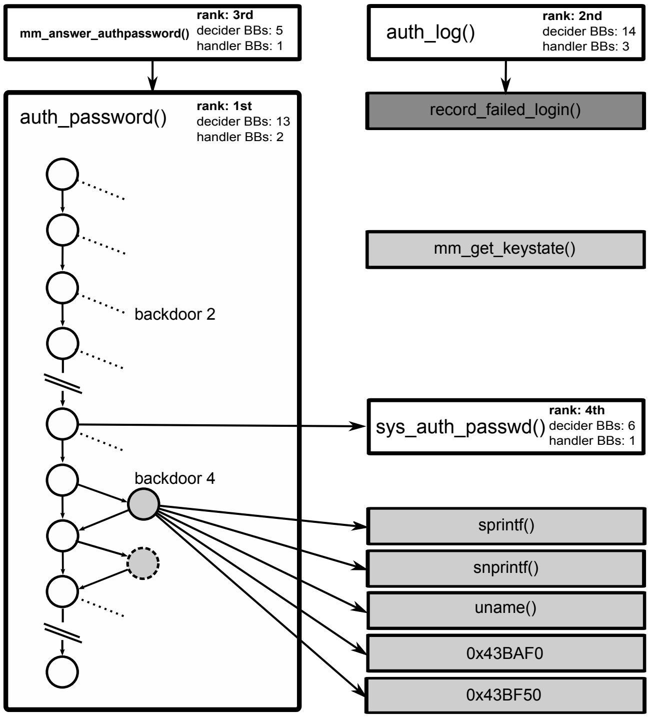


Figure 4: Decision tree for the password authentication in the monitor process of the malicious version of OpenSSH. Functions are rectangles, basic blocks are circles. Deciders are white, handlers for *valid-pw* are gray, handlers for *invalid-pw* are dark-gray. Cold edges are dotted.

sion tree of the slave process only contains exclusive handler functions for the protocol runs of the commands *password*, *publickey* and *none*: `41BF00h` (`userauth_passwd()`), `41CA50h`(`userauth_pubkey()`) and `41BE20h` (`userauth_none()`), respectively. These results suggests that the *hostbased* authentication method is disabled – a circumstance that can be verified by manual analysis. Based on this results, Weasel's analysis module that heuristically scans for function pointer tables as described in Section 4.2 automatically and unambiguously identifies the correct address and size of the command descriptor table in the `.data` section of the binary file, spanning from virtual address `674678h` to `6746f8h`. The table, which is defined in the OpenSSH source code under the identifier `authmethods`, contains simple structures of three members (*name*, *handler function*, *enabled flag*) describing all available authentication methods of the server. Interestingly, the analysis module identifies two handler functions not contained in any of the collected traces: `41BD40h`(`userauth_kbdint()`) and `41B9F0h` (`userauth_hostbased()`). While the latter belongs to the disabled authentication method *hostbased*, the former belongs to the well-known authentication method *keyboard-interactive* which is not described in RFC 4252 (and suffers as well from backdoors X2 and X4). This demonstrates the ability of Weasel to identify handlers for unknown commands. The decision tree of the monitor process contains only the monitor-side handler function for the *password* authentication method. We do not discuss it further due to space restrictions.

5.1.2 Dropbear SSH (MIPS32)

We examined a binary-only version of Dropbear SSH server shipped as part of the firmware of the Siemens VoIP desk telephone *OpenStage 40*. Function and basic block traces were recorded remotely on the embedded hardware. We were able to automatically and unambiguously identify important deciders and handlers in both the authentication and the command dispatching process of the SSH server.

Authentication.

The rather small decision tree computed from the *valid-pw* and *invalid-pw* protocol runs is depicted in Figure 5.

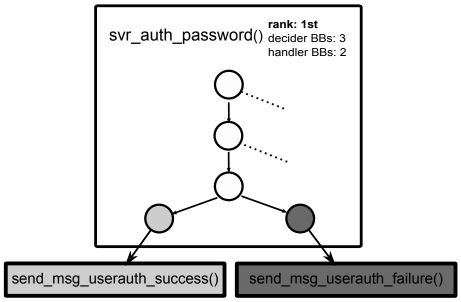


Figure 5: Decision tree of the authentication process of Dropbear SSH.

The only identified decider `svr_auth_password()` evaluates the correctness of a password by a simple string comparison and, depending on the outcome, subsequently calls one of the two identified handlers. No suspicious external functions are reachable from either handler.

Command Dispatching.

The decision tree computed from the protocol runs corresponding to the four authentication methods of the SSH-AUTH protocol contains three deciders, with `recv_msg_userauth_request()` ranking first. From this decider, the only exclusive handlers `svr_auth_password()` and `svr_auth_pubkey()` are called, which belong to the *password* and the *publickey* authentication method, respectively. The application does not contain exclusive handlers for the other authentication methods. As Dropbear SSH in general does not dispatch commands via function pointer tables, Weasel does in this case correctly not recognize any function pointer tables of interest. The two identified exclusive handler functions were not found to lead to any suspicious calls.

5.2 ProFTPD (x86, x64, MIPS32)

Graduate students from our group not related to this project were asked to implement an arbitrary set of backdoors for ProFTPD in addition to the real-world backdoor of our running example. We have chosen ProFTPD since it is a complex program (e.g., FTP is a non-trivial protocol, it contains different software modules, and there were real-world attacks against this program) but still has a manageable code base. Altogether, eleven different backdoors were developed in a not supervised manner.

Seven out of these eleven backdoors interfere with the authentication or command dispatching process of ProFTPD and can thus theoretically be identified using Weasel. The other four backdoors implement malicious functionality like a covert out-of-band interactive shell that cannot be found by applying the described detection approach. We thus evaluate Weasel on the following set of backdoors containing the seven artificial backdoors and our running example:

Y1 *Acidbitchez*: our real-world running example.

Y2 *Happy hour*: at a certain time of day all passwords are accepted by the server.

Y3 *Blessed one*: for a certain client IP address all passwords are accepted.

Y4 *File access*: hidden commands for unauthorized file access bypassing the authentication process.

Y5 *Credentials stealing 1*: validated credentials are stored and made available via a hidden command.

Y6 *Credentials stealing 2*: validated credentials are sent via DNS requests to a remote server.

Y7 *Self-modifying authentication*: in case of a certain process ID, the central authentication function is rewritten on startup of the application to accept arbitrary passwords.

Y8 *Authentication module*: a malicious auth. module.

We analyzed a version of ProFTPD containing this backdoors on Weasel's supported platforms x86, x64 and MIPS-32. Since the high-level results were identical for all platforms, we discuss them together (concrete numbers apply at least to the x64 version). We demonstrate that it is possible to reliably detect or to disable backdoors Y1–Y6, substantiating our approach to reduce the attack surface.

We cannot cope with backdoors Y7 and Y8: Backdoor Y7 can only be detected when during testing by chance a case is encountered where the authentication function is actually being overwritten (i.e., the backdoor triggers on a certain process ID). Backdoor Y8 can not be identified as Weasel currently does not evaluate dynamically loaded modules.

Authentication.

The decision tree computed from the *valid-pw* and *invalid-pw* protocol runs is depicted in Figure 6. On the function level, four deciders are identified, with `auth_pass()` with 78 deciders on basic block level, 25 handler functions for *valid-pw* and the 1st scoring rank being clearly dominant. Though, three of the eleven backdoors are located in the decider function `pr_auth_authenticate()`, which only leads to a single handler function (`do_back()`) for *valid-pw*. This handler function belongs to backdoor Y6 and stores credentials once they are successfully validated. The function can automatically be identified as highly suspicious, as it (among other activities) statically calls the standard functions `mmap()`, `shm_open()`, `socket()`, and `sendto()`. Two of the nine cold decider basic blocks in `pr_auth_authenticate()` lead to the implementations of backdoors Y2 and Y3, respectively. Thus cutting cold edges in the identified deciders would effectively render these backdoors non-functional.

In the x86 version of our modified ProFTPD server the code implementing the "happy hour" backdoor (Y2) in `pr_auth_authenticate()` uses the conditional instruction `CMOVZ` to manipulate the outcome of the function. This underlines the need to consider *implicit edges* when examining a function's CFG since we would otherwise overlook this case.

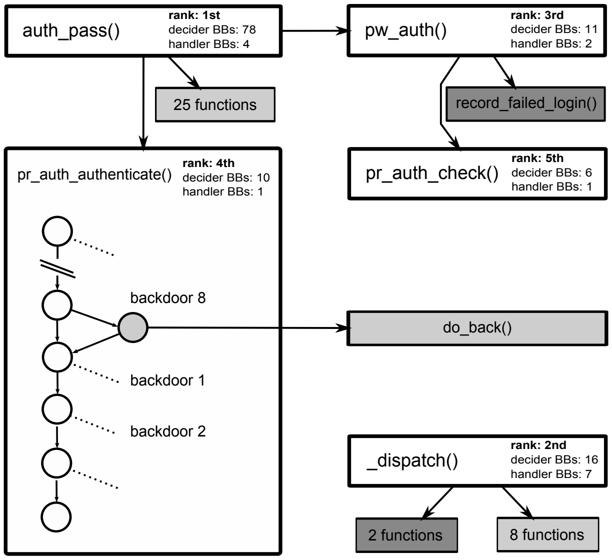


Figure 6: Decision tree of the authentication process of ProFTPD.

Command Dispatching.

Weasel's protocol description of FTP was modelled according to RFC 959 [29] and contains 34 commands (e.g., `HELP` and `MKD`). The function level decision tree consists of six deciders (of which `pr_cmd_dispatch_phase()` ranks 1st) and 60 handlers. Out of those handlers, 43 are exclusive to a single protocol run. By manual analysis it can be verified that these exclusive handlers indeed implement each of the 34 commands. For the majority of commands there exists exactly one exclusive handler. Subsequently, it is easily possible to automatically identify the backdoor of our running example (Y1): among the external functions reachable from `HELP` command's only exclusive handler `core_help()` in the static CFG are the aforementioned in this context highly suspicious ones `setegid()`, `seteuid()` and `system()`. Accordingly, the corresponding `HELP` command can be identified as suspicious and further defensive measures can be applied.

Weasel automatically and correctly identifies the addresses and sizes of exactly five function pointer tables in the address space of the respective ProFTPD process: `core_cmdtab`, `xfer_cmdtab`, `auth_cmdtab`, `ls_cmdtab`, and `delay_cmdtab`. The first one is the largest and contains 35 entries describing the core set of the commands supported by ProFTPD. Eight entries in `core_cmdtab` contain function pointers that are not contained in any of the recorded traces. By examining the respective entries in the table in the binary program, they can already be identified to be corresponding to the following commands: `EPRT`, `EPSV`, `MDTM`, `SIZE`, `DOWNLOAD`, `UPLOAD`, `GSM` and `RSLV`. While the first four are known benign FTP commands that are simply not defined in RFC 959, the last four belong to the backdoors Y4, Y5 and Y6. The other four identified function pointer tables also partly contain pointers to functions that were not encountered during testing. These functions correspond either to known commands (e.g., `xfer_log_stor()`) or the benign FTP command `PROT`, which is not defined in RFC 959 as well.

6. CONCLUSIONS

In this paper, we presented an approach towards the automatic detection and disabling of certain types of backdoors in server applications by carefully examining runtime traces for different protocol runs. Our implementation of the approach in the form of a tool called WEASEL automatically captures these traces by repeatedly invoking a server application under test according to a formal, block-based specification of the respective protocol. As WEASEL only relies on `gdbserver` for the recording of traces, it is widely applicable to a variety of platforms and we discussed several empirical analysis results that demonstrate how WEASEL can be used to precisely detect relevant code parts and data structures within a given binary application.

7. ACKNOWLEDGMENTS

We thank Andreas Maaß and Martin Steegmanns for implementing the discussed artificial backdoors, Moritz Contag for assisting with the development of our tool, and the anonymous reviewers for helpful comments. This work has been supported by the German Federal Ministry of Education and Research (BMBF) under support code 16BP12302; EUREKA-Project SASER.

8. REFERENCES

[1] GDB Remote Serial Protocol. `http://sourceware.org/gdb/onlinedocs/gdb/Remote-Protocol.html`.

[2] ProFTPD Backdoor Unauthorized Access Vulnerability, 2010. `http://www.securityfocus.com/bid/45150`.

[3] Backdoor Found In Arcadyan-based Wi-Fi Routers, 2012. `http://it.slashdot.org/story/12/04/26/1411229/backdoor-found-in-arcadyan-based-wi-fi-routers`.

[4] RuggedCom - Backdoor Accounts in my SCADA network? You don't say..., 2012. `http://seclists.org/fulldisclosure/2012/Apr/277`.

[5] Samsung printers contain hidden, hard-coded management account, 2012. `http://www.zdnet.com/samsung-printers-contain-hidden-hard-coded-management-account-7000007928/`.

[6] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar. Trojan Detection using IC Fingerprinting. In *IEEE Symposium on Security and Privacy*, 2007.

[7] D. Aitel. An Introduction to SPIKE, the Fuzzer Creation Kit. `www.blackhat.com/presentations/bh-usa-02/bh-us-02-aitel-spike.ppt`, 2002. Presented at Black Hat US.

[8] P. Amini and A. Portnoy. Fuzzing Sucks! Introducing Sulley Fuzzing Framework. `pentest.cryptocity.net/files/fuzzing/sulley/introducing_sulley.pdf`, 2007. Presented at Black Hat US.

[9] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-privilege Compartments. In *USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[10] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. X. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In W. Lee, C. Wang, and D. Dagon, editors, *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 65–88. Springer, 2008.

[11] D. Brumley and D. Song. Privtrans: automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, 2004.

[12] S. Dai, T. Wei, C. Zhang, T. Wang, Y. Ding, Z. Liang, and W. Zou. A framework to eliminate backdoors from response-computable authentication. In *IEEE Symposium on Security and Privacy*, 2012.

[13] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: preventing authentication & access control vulnerabilities in web applications. In *USENIX Security Symposium*, 2009.

[14] L. Duflot. CPU Bugs, CPU Backdoors and Consequences on Security. In *European Symposium on Research in Computer Security (ESORICS)*, 2008.

[15] S. Duquette. Linux/SSHDoor.A Backdoored SSH daemon that steals passwords, jan 2013. `http://www.welivesecurity.com/2013/01/24/linux-sshdoor-a-backdoored-ssh-daemon-that-steals-passwords/`.

[16] H. Flake. Structural comparison of executable objects. In *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2004.

[17] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, pages 238–255. Springer, 2008.

[18] D. Geneiatakis, G. Portokalidis, V. P. Kemerlis, and A. D. Keromytis. Adaptive defenses for commodity software through virtual application partitioning. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[19] Y. L. Gwon, H. T. Kung, and D. Vlah. DISTROY: detecting integrated circuit Trojans with compressive measurements. In *USENIX Workshop on Hot Topics in Security*, 2011.

[20] J. S. Havrilla. Borland/Inprise Interbase SQL database server contains backdoor superuser account with known password, 2001. `http://www.kb.cert.org/vuls/id/247371`.

[21] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *IEEE Symposium on Security and Privacy*, 2010.

[22] D. Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, 2003.

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

[24] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2008.

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

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[27] H. D. Moore. Shiny Old VxWorks Vulnerabilities, 2010. `https://community.rapid7.com/community/metasploit/blog/2010/08/02/shiny-old-vxworks-vulnerabilities`.

[28] D. G. Murray and S. Hand. Privilege separation made easy: trusting small libraries not big processes. In *European Workshop on System Security (EuroSec)*, 2008.

[29] J. Postel and J. Reynolds. File Transfer Protocol. RFC 959 (INTERNET STANDARD), Oct. 1985. Updated by RFCs 2228, 2640, 2773, 3659, 5797.

[30] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, 2003.

[31] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Symposium on Network and Distributed System Security (NDSS)*, 2011.

[32] S. Sparks, S. Embleton, and C. C. Zou. A chipset level network backdoor: bypassing host-based firewall & ids. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2009.

[33] C. Sturton, M. Hicks, D. Wagner, and S. T. King. Defeating UCI: Building Stealthy and Malicious Hardware. In *IEEE Symposium on Security and Privacy*, 2011.

[34] M. Tehranipoor and F. Koushanfar. A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Design & Test of Computers*, 27(1), 2010.

[35] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8), Aug. 1984.

[36] A. Waksman and S. Sethumadhavan. Silencing hardware backdoors. In *IEEE Symposium on Security and Privacy*, 2011.

[37] C. Wysopal, C. Eng, and T. Shields. Static detection of application backdoors - detecting both malicious software behavior and malicious indicators from the static analysis of executable code. *Datenschutz und Datensicherheit*, 34(3):149–155, 2010.

[38] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252 (Proposed Standard), Jan. 2006.

[39] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Trans. Comput. Syst.*, 20(3):283–328, Aug. 2002.

[40] A. Zeller. Isolating cause-effect chains from computer programs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2002.

Appendix A. A-WEASEL ALGORITHM

Given a set of n traces on function level the A-WEASEL algorithm recursively performs the following steps:

1 Determine the set of functions present in all n traces:

$$S_{common,funcs} = S_{T_0,funcs} \cap S_{T_1,funcs} \cap ... \cap S_{T_{n-1},funcs}$$

2 For each trace, determine the set of exclusive functions:

$$S_{T_i,ex,funcs} = S_{T_i,funcs} \setminus S_{common,funcs}$$

3 For each exclusive function in each set $S_{T_i,ex,funcs}$, determine the minimum number of call stack levels needed to distinguish between all of its invocations in the call graph (CG) of T_i. We denote the minimum call stack needed for distinction the *signature call stack* of an invocation. A new set $S_{T_i,ex,funcs,callstack}$ containing all invocations with different signature call stacks of all functions in $S_{T_i,ex,funcs}$ is created:

$$S_{T_i,ex,funcs,callstack} = \delta(S_{T_i,ex,funcs})$$

4 For each set $S_{T_i,ex,funcs,callstack}$, remove those invocations of exclusive functions from the set that are *dominated* by other exclusive functions in the call graph of the corresponding trace T_i:

$$S_{T_i,ex,funcs,top} = \varphi(S_{T_i,ex,funcs,callstack})$$

Thus only top-level invocations of exclusive functions of T_i are contained in the set.

5 Group all remaining invocations in all sets $S_{T_i,ex,funcs,top}$ according to their signature call stacks. Invocations with compatible signature call stacks are grouped together. Two signature call stacks are compatible if both are equal or are equal up to the end of one of the two call stack. Note how each group only corresponds to one specific exclusive function and can at most contain one specific invocation from each trace.

6 The immediate parent function in the common call stack of a group's exclusive function is necessarily in set $S_{common,funcs}$ and is added as *decider* to the *decision tree*. Note that several groups can also share a common decider function. In case a group consists of only a single invocation from a single trace, the corresponding exclusive function is added as *handler* to the decision tree. Recursion ends in this case.

7 For each group, dynamically trace the corresponding *decider* function for the group's common signature call stack for all applicable protocol runs on basic block level. From the recorded basic block traces, the internal *decision tree* of the decider function for the signature call stack is generated by a similar but simpler algorithm.

8 For each group recursively execute A-WEASEL. For each invocation belonging to the group, a self-contained and minimal sub-trace T'_i is cut from the original trace T_i that starts with the corresponding signature call stack. A-WEASEL is executed on the set of all such sub-traces corresponding to the group. I.e., A-WEASEL is executed on the sub-CGs of the *decider* identified for the group for all applying traces.

9 In case a *decider* function is found to be a leaf in the resulting decision tree and does not exhibit any control flow differences on basic block level for applicable protocol runs, it is transformed into a common *handler* function.