Towards Semi-automated Detection of Trigger-based Behavior for Software Security Assurance

Dorottya Papp
CrySyS Lab, Dept. of Networked Systems and Services, BME
Magyar tudósok krt 2.
Budapest 1117, Hungary
Center of Digital Safety and Security, Austrian Institute of Technology
Donau-City-Straβe 1.
Vienna, Austria
dpapp@crysys.hu

Levente Buttyán
CrySyS Lab, Dept. of Networked Systems and Services, BME
Magyar tudósok krt 2.
Budapest 1117, Hungary
buttyan@crysys.hu

Zhendong Ma
Center of Digital Safety and Security, Austrian Institute of Technology
Donau-City-Straβe 1.
Vienna, Austria
zhendong.ma@ait.ac.at

ABSTRACT

A program exhibits trigger-based behavior if it performs undocumented, often malicious, functions when the environmental conditions and/or specific input values match some pre-specified criteria. Checking whether such hidden functions exist in the program is important for increasing trustworthiness of software. In this paper, we propose a framework to effectively detect trigger-based behavior at the source code level. Our approach is semi-automated: We use automated source code instrumentation and mixed concrete and symbolic execution to generate potentially suspicious test cases that may trigger hidden, potentially malicious functions. The test cases must be investigated by a human analyst manually to decide which of them are real triggers. While our approach is not fully automated, it greatly reduces manual work by allowing analysts to focus on a few test cases found by our automated tools.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**; *Software security engineering*; • **Software and its engineering → Operational analysis**;

KEYWORDS

Mixed Concrete and Symbolic Execution, Trigger-based Behavior, Static Analysis, Source Code Analysis, Software Security

ACM Reference format:
Dorottya Papp, Levente Buttyán, and Zhendong Ma. 2017. Towards Semi-automated Detection of Trigger-based Behavior for Software Security Assurance. In *Proceedings of ARES '17, Reggio Calabria, Italy, August 29-September 01, 2017,* 6 pages.
https://doi.org/10.1145/3098954.3105821

ARES '17, August 29-September 01, 2017, Reggio Calabria, Italy

ACM ISBN 978-1-4503-5257-4/17/08…$15.00
https://doi.org/10.1145/3098954.3105821

1 INTRODUCTION

Trigger-based behavior in software refers to execution of code that is activated on specific input values (the so called *trigger inputs*) and that performs some undocumented functions. Often, the users are not aware of the existence of those undocumented functions in the software. It is clear that using an application that has such hidden functions, which can be triggered by non-legitimate parties, is dangerous, as those functions can potentially implement malicious actions. Indeed, the best examples for trigger-based behavior include *backdoors* and *logic bombs* hidden in applications by malicious parties [25], although there are benign examples as well, such as *easter eggs*. Malware can also exhibit trigger-based behavior [9, 17, 19]; however, in this paper, we are more interested in trigger-based behavior in legitimate applications.

The adversary model of trigger-based behavior assumes that the attacker has access to the software and is able to modify the implementation to implant the hidden, malicious behavior, which will be triggered during the execution of the software. In practice, the attacker may be a compromised developer or a compromised contractor in the supply-chain trying to insert malicious code in the software, which can be exploited later in a targeted attack against the user [6]. Or, the attacker may be a disgruntled employee at a company with access to the software used by that company [14].

From a software security assurance perspective, the detection of trigger-based behavior in software is of paramount importance. Unfortunately, it is a difficult task. One approach is to try detecting hidden functions by software testing. However, current software testing approaches have limitations. Specification-based (or black-box) testing techniques have no knowledge of the internal structure of the software, therefore, they may only discover the hidden behavior if the correct trigger inputs are specified. This limitation reduces their effectiveness because of the assumption that no one, but the attacker has knowledge of the trigger inputs. Structural (or white-box) testing, on the other hand, takes into account the internal structure of the software which the hidden behavior is part of. As such, white-box testing approaches can generate better results.

White-box analysis can be performed at the binary level or at the source code level. However, analyzing the binary can be difficult due to the lack of semantic abstractions in low level code. And

even if one has access to the source code, detecting trigger-based behavior by manual inspection can be tedious and error prone, as the human analyst has to interpret a potentially large amount of code written by someone else. Yet, today, manual source code analysis seems to be the approach that is used by practitioners [1]. Although some researchers tried to automate the detection of trigger-based behavior, most of those works focus on binary analysis of malware (see e.g., [2]), in which we are not interested in this work.

In this paper, we apply open-source and commercially available tools to effectively detect trigger-based behavior at the source code level. We propose a framework which utilizes existing white-box analysis tools. Our approach is based on automated source code instrumentation that makes the source code amenable to analysis by tools. Detection of trigger-based behavior is achieved by mixed concrete and symbolic execution, which automatically outputs numerous test cases for uncovered execution paths in the analyzed software. Among those test cases, a small subset of potentially suspicious test cases are automatically highlighted by our tools. Those highlighted test cases must be investigated by a human analyst to decide which of them could be a trigger input. Thus, our approach is not fully automated, but it greatly reduces manual work by allowing analysts to focus on the few highlighted test cases found by our automated tools. More specifically, our contributions are the following:

(1) we present a framework for semi-automated detection of trigger-based behavior based on existing tools. To the best of our knowledge, we are the first to show that existing mixed concrete and symbolic execution tools can be successfully deployed for detecting trigger-based behavior;
(2) we demonstrate a proof-of-concept implementation of the framework for programs written in C using the LLVM toolchain;
(3) we evaluate the results on real-world, open-source software samples. Our first result suggests that if mixed concrete and symbolic execution succeeds, the hidden, malicious behavior can be detected.

The paper is structured as follows. The challenges of detecting trigger-based behavior are discussed in Section **??**, together with existing approaches to overcome those challenges. We present our framework in Section 3. Our preliminary results of applying the framework to real-life, open-source examples are presented in Section 4. Finally, Section 5 concludes the paper and sketches future work.

2 CHALLENGES OF DETECTING TRIGGER-BASED BEHAVIOR AND EXISTING APPROACHES

The challenges of detecting trigger-based behavior mainly arise from the fact, that the analysis has to uncover a hidden, stealthy type of behavior which is executed only under very specific circumstances. Black-box testing approaches observe only the inputs and outputs of the software and have no knowledge of the internals of the software. Therefore, the hidden behavior can only be detected, if the analyst either knows the trigger inputs beforehand or can correctly guess them. Considering the threat model, it is unreasonable to assume that the tester knows the trigger inputs. In addition, the probability of correctly guessing the trigger inputs is low.

White-box testing approaches, on the other hand, take into account the internal structure of software. However, current automated vulnerability-finding tools tend to focus on typical programming vulnerabilities that are exploited by well-known and understood attacks (e.g. buffer overflow). To assist analysts in creating test cases that cover trigger-based behavior as well, the SQA Tool [1] highlights code segments based on how much test-coverage would improve with their execution. This approach certainly improves the performance of human analysts, but the detection process is not automated enough. Test cases must still be written by the analysts, which requires interpretation of the source code, as well as correctly determining of trigger inputs based on the source code.

Taking into account the internal structure of software is useful for the detection of trigger-based behavior, as shown in [8]. The authors modeled malicious behavior observed during execution, and were able to detect similar code segments in new malware samples. However, this approach relies on already observed behavior and cannot detect new types of trigger-based behavior.

The ideal approach for detecting trigger-based behavior should be able to:

(1) interpret how input values are handled, which is analogous to how input values modify the behavior of the analyzed software,
(2) automatically generate input values based on the interpretation,
(3) identify conditions required to reach any part of the code,
(4) detect suspicious conditions and decide whether the execution path is malicious or benign

The first three requirements can be satisfied with symbolic execution [5], a technique that is able to calculate the constraints on inputs such that execution takes a certain path (also known as the path condition). The technique assigns symbolic values to variables, that describe and track how the value of the variables depend on the input. At branches in the software, symbolic execution splits into two instances: in one instance, the symbolic variables are conditioned so that they satisfy the branching condition, in the other, they do not. Solving the path condition results in concrete input values that can be used as test cases covering previously uncovered code. Solving the path condition is typically performed by a decision procedure (also known as the solver) such as Yices [11], Z3 [10] or STP [13].

However, traditional symbolic execution suffers from a number of limitations. Here we only summarize these limitations, interested readers are directed to the study presented in [5].

- Path explosion: by splitting execution at each branch, the number of program paths to be explored usually grows exponentially
- Environment problem: all code that resides outside of the analyzed software, but is used by it, should be executed symbolically as well for precise results, including all libraries and the operating system

- Resource constraint: constraint solving takes up much of the time during symbolic execution and hinders the technique's scalability, moreover, splitted execution instances require much memory
- Unsolvable conditions: some conditions cannot be solved by the solver (e.g. matching the cryptographic hash value of the symbolic input to a hard-coded one)

To overcome some of the challenges, a hybrid approach was proposed that mixes concrete and symbolic execution [4, 15, 21]. Analysis maintains two states: one maps variables to their concrete values, while the other maps symbolic variables. As execution progresses, both states are updated with the new concrete values and the new constraints on symbolic values. The advantage of this approach is that even if the solver fails, the state that maps variables to concrete values can be used to explore at least one of the two paths emanating from the branch. In addition, the stored concrete values cane be used when interacting with external code, thereby solving the environment problem.

The modern approach of mixing concrete and symbolic execution has been proposed for the detection of trigger-based behavior in literature. Minesweeper [2], for example, utilizes this technique together with dynamic binary instrumentation and has been shown to successfully identify trigger-based behavior in real-world malware in binary form. Triggerscope [12] also relies on mixed concrete and symbolic execution but it focuses on Dalvik bytecode instead of native binaries. Rozzle [16] maintains symbolic values for JavaScript variables dependent on environmental-specific values and splits execution at branches to explore multiple program paths.

The above mentioned related works resulted in the implementation of new mixed concrete and symbolic execution engines. However, there exists already many open-source or commercially available mixed concrete and symbolic execution tools, such as KLEE [4] and CUTE [21] on the source code level, Triton [20] and Angr [22] on the binary level. Our approach is different from previous work in that we study how *existing tools* could be used for detecting trigger-based behavior. The rationale of our approach is that mixed concrete and symbolic execution is an active research area and improvements in the field will lead to better tools in the future. Therefore, it is advantageous to have an approach in which tools can be interchanged with minimal overhead.

3 OUR APPROACH

In this section, we present our approach to detect trigger-based behavior using existing mixed concrete and symbolic execution tools. Our approach works at the source code level and is capable of both detecting trigger-based behavior and supporting human analysts by outputting the instructions leading to the suspicious behavior. The high-level overview of our proposed framework is shown in Fig. 1.

3.1 Automatic Source Code Instrumentation

Many existing mixed concrete and symbolic execution tools, including KLEE [4], S2E [7] and CREST [3], require additional library calls to specify which variables in the software should be treated as symbolic. Therefore, the first step of the analysis is to instrument the source code with the required library calls.

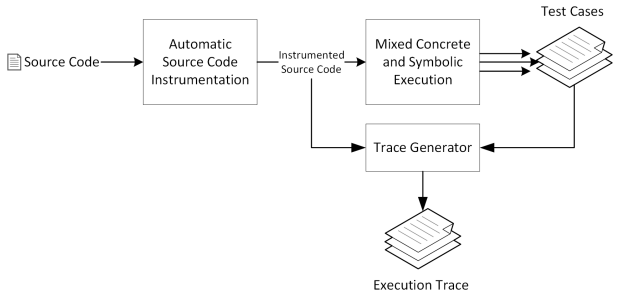

Figure 1: High-level Overview of the Framework

In order to detect trigger-based behavior, instrumentation must first identify variables and function calls that may guard the hidden behavior. Considering that the hidden behavior is triggered by the attacker during execution, the trigger inputs must be supplied via interaction with the software, thus, the attacker is part of the environment. As software typically interacts with its environment via function calls, calls that return data from the environment are potential entry points for trigger inputs. By replacing these calls so that fresh symbolic values are returned, mixed concrete and symbolic execution can determine how execution depends on the environment.

We propose that replacement should happen with *dummy functions*. Dummy functions are empty functions, they do not implement any functionality, only introduce fresh symbolic values to the software under analysis. They have the same prototype as their original counterpart (i.e. same return type, same number and type of arguments). The introduction of fresh symbolic values can happen either as a return value or by making certain arguments symbolic, depending on the semantics of the original function. Dummy functions should be easily identifiable in the source code, e.g. with naming convention, so that the introduction of symbolic values is obvious. For example, consider the `pcap_next()` function from the libpcap library. The function reads the next packet from a packet capture, and as such, it may return the trigger inputs embedded in a packet. The original function has the signature `const u_char *pcap_next(pcap_t*, struct pcap_pkthdr*)`. The corresponding dummy function has the same return type and arguments, but it is easily identified with a naming convention: `const u_char *pcap_next_dummy(pcap_t*, struct pcap_pkthdr*)`. In addition, instead of returning a pointer to a concrete packet, the dummy function returns a pointer to a symbolic value representing a packet.

3.2 Mixed Concrete and Symbolic Execution

The instrumented source code is then analyzed by the mixed concrete and symbolic execution tool which is capable of automatically generating test cases for the analyzed software. To increase the effectiveness of our approach, we require the following two features:

(1) The tool must be able to output path conditions at any given program point
(2) Path conditions output at potentially malicious points must be easily differentiated from other test cases generated at different program points

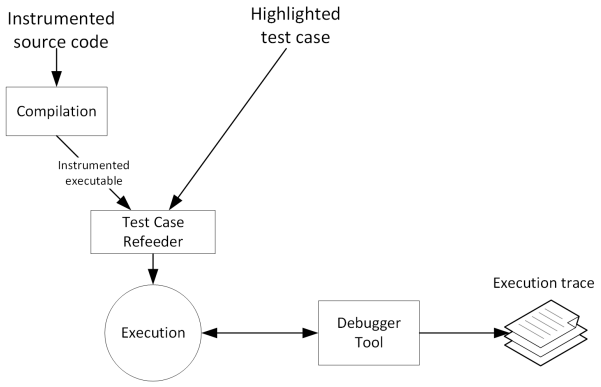

Figure 2: Trace Generation

The first feature is used to output path conditions before potentially dangerous instructions. We define potentially dangerous instructions as system and library calls that could be used for implementing malicious behavior. For example, the `execv()` call in C is potentially dangerous because it may be used to give malicious commands to the operating system. Other potentially dangerous instructions include calls to `system()` and `send()`. Given a list of potentially dangerous functions, their use in the source code can be identified automatically. Thus, the source code can be automatically instrumented in such a way, that the mixed concrete and symbolic execution tool generates a test case immediately before reaching the suspicious behavior. Since potentially dangerous system and library calls can be used for benign purposes as well, false positives are possible.

The second requirement increases the level of support our approach gives to human analysts by prioritizing them. We refer to solutions to path conditions output at potentially malicious program points as *highlighted test cases*. Highlighting potentially malicious test cases orients analysts towards potential trigger inputs. The differentiation between ordinary and highlighted test cases may be based on special file extensions, storage in a different folder, etc.

3.3 Trace Generation

Highlighted test cases show the potential trigger inputs, but give no information about how the trigger inputs are used in the software. This information is acquired by trace generation. The idea here is to replay the highlighted test cases to see the sequence of instructions executed (the trace) and follow the same execution path, that the mixed concrete and symbolic execution tool did. The overview of trace generation is shown in Fig. 2.

Firstly, the instrumented source code has to be compiled into an executable which contains the dummy functions. The compiled software is then executed in the test case refeeder, which also takes as input the highlighted test case. The task of the refeeder is to monitor the execution of the instrumented software and replace symbolic values with the concrete values from the highlighted test case. Whenever dummy functions would introduce fresh symbolic values, the test case refeeder intercepts the call and returns the input value calculated by the mixed concrete and symbolic execution tool. Meanwhile, a debugger is attached to the process to generate the trace. The debugger steps through the software line by line and outputs each line into the execution trace. The execution trace can then be inspected by human analysts to determine whether the execution path is indeed malicious or not.

Table 1: Tools Used in Prototype Implementation

Framework Element	Tool in Prototype
Automatic source code instrumentation	Clang [18]
Mixed concrete and symbolic execution	KLEE
Compilation	GCC [24]
Test case refeeder	klee-replay
Debugger	GDB [23]

3.4 Prototype Implementation

We implemented our approach as a prototype using the GNU and LLVM toolchains. The tools are summarized in Table 1. Automatic source code instrumentation was implemented as a standalone tool using clang. Our prototype automatically generates dummy functions based on a list of function names and semantic data about how to introduce fresh symbolic values. The following JSON document shows the semantic data required to generate dummy functions:

```
{
    "function_name"         : "recv",
    "include_file"          : "sys/socket.h",
    "symbolic_return"       : false,
    "symbolic_return_size"  : 0,
    "symbolic_params"       : [
        {
            "index"             : 1,
            "has_fixed_length"  : false,
            "fixed_length"      : 0,
            "has_dynamic_length" : true,
            "length_param_index" : 2
        }
    ]
}
```

The JSON document contains the name of the original function and the header file from which it is included. It also encodes whether the return value of the dummy function should be made symbolic (`"symbolic_return"`) and the size of the symbolic return value (`"symbolic_return_size"`). There function calls, however, that write the environment-specific data into one of their parameters, such as `recv()`, which writes the message from the socket into its second parameter, a buffer. Therefore, the structure encodes whether any of the parameters should be made symbolic (`"symbolic_params"`). For each such parameter, its index is given (starting from 0) with additional information about how to create the symbolic value. Some function calls, like `pcap_next()`, allocate the buffer themselves based on the environment-specific data and return a pointer to it. In such cases, our prototype implementation generates a symbolic value with fixed length (`"has_fixed_length"`). The fixed length is a numeric value, e.g. 35 (`"fixed_length"`). There are function calls, however, which also expect a numeric

parameter giving the upper limit of data size the buffer can hold. In such cases, our prototype generates a symbolic value with dynamic length (`"has_dynamic_length"`) using the index of the parameter holding the size limit (`"length_param_index"`).

Our prototype uses KLEE as a mixed concrete and symbolic execution tool, because KLEE satisfies the two feature requirements mentioned in Section 3.2. Our implementation signals KLEE to output highlighted test cases using the `klee_assert()` function. The function takes a logical formula as input and is intended for sanity checks. Failing sanity checks result in the termination of the current execution path and KLEE outputs the uncovered path condition with its solution (if it can be computed). We signal potentially dangerous program points by giving the function an always failing logical formula and in this way, forcing KLEE to calculate and output the potential trigger inputs. Currently, the `klee_assert()` call is placed in the source code before the following functions:

- `system()` and `exec()`, as they can be used to give commands to the operating system, and
- `send()`, as it can be used to leak information about the system.

The list can be extended to include more potential malicious functions, for example, from `unistd.h` and `socket.h` headers.

The instrumented source code is compiled with GCC and fed to klee-replay, a replay library provided by KLEE. The replay library also takes as input the test cases generated by KLEE, which have the KTEST extension. Sanity check failures result in outputting not only the generated test case and the path condition, but also a text-based file named `test<numerical ID>.external.err`. This special file extension highlights potential trigger inputs. The contents of the binary file can be read with the ktest-tool utility, which lists the concrete values deducted from the path condition:

```
object 1: name: 'arg0'
object 1: size: 11
object 1: data: '-X\x00\x00\x00\x00\x00\x00\x00\x00\x00'
```

During replay, gdb is connected to the process in which the instrumented software is executed. Using the `step` and `next` gdb commands, the source code lines executed are outputted into a file for further inspection.

4 PRELIMINARY RESULTS

To evaluate our approach, we collected open-source software from GitHub using keyword search for the terms “backdoor”, “logic bomb”, “time bomb” and “portknock”. All collected samples were written in C and implement some form of trigger-based behavior. For the evaluation, we used a virtual machine with 4 CPUs and 10 GB memory. The virtual machine ran Ubuntu 14.04.5 LTS. We set the maximum memory available to KLEE to 8 GB.

Our results are summarized in Table 2. While KLEE explored many paths in the samples, we configured it in such a way, that only test cases covering previously uncovered code would be outputted. Hence the low number of test cases generated.

The cd00r project uses a filtered packet capture and starts an interactive shell after a successful portknock. This sample executes in an infinite loop and exits only, if portknocking is successful. Because of the infinite loop, however, our analysis would have taken an infinite amount of time as well. Therefore, we modified the code so that unsuccessful attempts cause it to exit as well. With this modification, KLEE generated 5 test cases of which 1 was highlighted. The highlighted test case was a true positive detection, and the only malicious path in the sample: no false negative test case was generated.

The giardia project expects a password to be delivered to the correct port. The password is configurable, in the original code, it is "s3cr3t", which we did not change. In this case, KLEE generated 4 test cases with 1 highlighted. The highlighted test was a true positive detection, and the tool did not miss any malicious paths.

The project osx-ping-backdoor was written for the OSX operating system, while our prototype implementation ran on Ubuntu. Therefore, we copied the malicious logic from the OSX implementation and injected it into the Ubuntu-compatible source code of the ping command. The malicious logic introduces two undocumented commandline parameters (`-x` and `-X`), both leading to the same code segment. 122 test cases were generated with 1 highlighted. The highlighted test case was a true positive detection, but KLEE missed the other commandline parameter. After the creation of the first successful highlighted test case, the second call to `klee_assert()` failed and the tool abandoned the execution path without outputting any results.

Analysis of the portknockd project failed because the maximum of 8 GB memory was not enough for KLEE. When the memory limit was exceeded, the tool abandoned thousands of paths, including the one implementing the hidden behavior.

The portknocking functionality in the portknocking project is protected by time: a portknocking attempt is considered successful only, if it happens within an small timing window. The timing window is implemented in a different thread, and so, the sample executes concurrently. However, KLEE was unable to analyze concurrent execution, and so, the analysis failed.

5 CONCLUSION AND FUTURE WORK

In this paper, we presented a framework for detecting trigger-based behavior utilizing automatic source code instrumentation, mixed concrete and symbolic execution and test case refeeding. Our framework automates most of the analysis and involves human analysts only during its latest phases. What is more, our approach works with existing tools which makes it both affordable and easily available to all developers.

The first results of the framework are promising: out of five real-life samples with trigger-based behavior, the framework correctly identified the hidden behavior in three of them. In the unsuccessful cases, the limitations experienced were caused by shortcomings of the chosen tool, not our approach.

Nevertheless, there are some improvements we list as future work. Firstly, our approach is currently capable of generating the execution trace, but the analysis of the trace left to the human analyst. We wish to amplify this approach by automatically analyzing the trace and generating a report on the behavior of the analyzed software at a higher semantic level. Therefore, interpretation of analysis results will be easier for the human analyst. We also wish to use the presented framework for a large scale analysis of open-source software and test them for trigger-based behavior.

Table 2: Samples and Results

Sample Name	Completed paths	Generated test cases (trigger inputs)	Detected
cd00r	1299	5 (1)	Yes (1/1)
giardia	48	4 (1)	Yes (1/1)
osx-ping-backdoor	212754	122 (2)	Yes (1/2)
portknockd	11902399	1	No
portknocking	39077	8	No

ACKNOWLEDGMENTS

The research leading to this paper has received funding from the H2020-ECSEL programme under the grant agreement no. 692474 (AMASS).

REFERENCES

[1] Hira Agrawal, James Alberi, Lisa Bahler, Josephine Micallef, Alexandr Virodov, Mark Magenheimer, Shane Snyder, Vidroha Debroy, and Eric Wong. 2012. Detecting Hidden Logic Bombs in Critical Infrastructure Software. In *International Conference on Information Warfare and Security*. Academic Conferences International Limited, 1.

[2] David Brumley, Cody Hartwig, Zhenkai Liang, James Newsome, Dawn Song, and Heng Yin. 2008. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*. Springer, 65–88.

[3] Jacob Burnim and Koushik Sen. 2008. Heuristics for scalable dynamic test generation. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. IEEE, 443–446.

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[5] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[6] DM Cappelli, AP Moore, RF Trzeciak, and Timothy J Shimeall. 2008. Common sense guide to prevention and detection of insider threat. *CERT Insider Threat Study Team, Carnegie Mellon University* (2008).

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.

[8] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. 2010. Identifying dormant functionality in malware programs. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 61–76.

[9] Cyber4Sight. 2017. Analysis of Malware Used Watering-Hole Attacks Against Polish, Other Financial Institutions. https://blog.cyber4sight.com/2017/02/technical-analysis-watering-hole-attacks-against-financial-institutions/. (Feb 2017).

[10] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[11] Bruno Dutertre. 2014. Yices 2.2. In *Computer-Aided Verification (CAV'2014) (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 737–744.

[12] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 377–396.

[13] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-vectors and Arrays. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*. Springer-Verlag, Berlin, Heidelberg, 519–531. http://dl.acm.org/citation.cfm?id=1770351.1770421

[14] Sharon Gaudin. 2008. IT Worker Jailed for Creating Logic Bomb. http://www.computerworld.com/article/2551740/government-it/it-worker-jailed-for-creating-logic-bomb.amp.html. (2008).

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 213–223. https://doi.org/10.1145/1065010.1065036

[16] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. 2012. Rozzle: De-cloaking Internet Malware. In *2012 IEEE Symposium on Security and Privacy*. 443–457. https://doi.org/10.1109/SP.2012.48

[17] Malwarebytes Lab. 2016. Shakti Trojan: Technical Analysis. https://blog.malwarebytes.com/threat-analysis/2016/08/shakti-trojan-technical-analysis/. (Aug 2016).

[18] Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.

[19] Atif Mushtaq. 2011. The Dead Giveaways of VM-Aware Malware. https://www.fireeye.com/blog/threat-research/2011/01/the-dead-giveaways-of-vm-aware-malware.html. (Jan 2011).

[20] Florent Saudel and Jonathan Salwan. 2015. Triton: A Dynamic Symbolic Execution Framework. In *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 31–54.

[21] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *ACM SIGSOFT Software Engineering Notes*, Vol. 30. ACM, 263–272.

[22] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*. 138–157. https://doi.org/10.1109/SP.2016.17

[23] Richard M. Stallman, Roland Pesch, and Stan et al. Shebs. 2011. *Debugging with GDB*. Free Software Foundation.

[24] Richard M. Stallman and the GCC Developer Community. 2013. Using the GNU Compiler Collection. https://gcc.gnu.org/. (2013).

[25] Chris Wysopal and Chris Eng. 2007. Static detection of application backdoors. *Black Hat* (2007).